

# Reasoning about Java’s Reentrant Locks

Christian Haack<sup>1\*</sup>, Marieke Huisman<sup>2\*†‡</sup> and Clément Hurlin<sup>3\*†</sup>

<sup>1</sup> Radboud Universiteit Nijmegen, The Netherlands

<sup>2</sup> University of Twente, The Netherlands

<sup>3</sup> INRIA Sophia Antipolis - Méditerranée, France

**Abstract.** This paper presents a verification technique for a concurrent Java-like language with reentrant locks. The verification technique is based on permission-accounting separation logic. As usual, each lock is associated with a resource invariant, i.e., when acquiring the lock the resources are obtained by the thread holding the lock, and when releasing the lock, the resources are released. To accommodate for reentrancy, the notion of lockset is introduced: a multiset of locks held by a thread. Keeping track of the lockset enables the logic to ensure that resources are not re-acquired upon reentrancy, thus avoiding the introduction of new resources in the system. To be able to express flexible locking policies, we combine the verification logic with value-parameterized classes. Verified programs satisfy the following properties: data race freedom, absence of null-dereferencing and partial correctness. The verification technique is illustrated on several examples, including a challenging lock-coupling algorithm.

## 1 Introduction

Writing correct concurrent programs, let alone verifying their correctness, is a highly complex task. The complexity is caused by potential thread interference at every program point, which makes this task inherently non-local. To reduce this complexity, concurrent programming languages provide high-level synchronization primitives. The main synchronization primitive of today’s most popular modern object-oriented languages — Java and C# — are reentrant locks. While reentrant locks ease concurrent programming, using them correctly remains difficult and their incorrect usage can result in nasty concurrency errors like data races or deadlocks. Multithreaded Java-like languages do not offer enough support to prevent such errors, and are thus an important target for lightweight verification techniques.

An attractive verification technique, based on the regulation of heap space access, is O’Hearn’s concurrent separation logic (CSL) [18]. In CSL, the programmer formally associates locks with pieces of heap space, and the verification system ensures that a piece of heap space is only accessed when the associated lock is held. This, of course, is an old idea in verification of shared variable concurrent programs [2]. The novelty of CSL is that it generalizes these old ideas in an elegant way to languages with unstructured heaps, thus paving the way from textbook toy languages to realistic programming languages. This path has been further explored by Gotsman et al. [10] and Hobor et

---

\* Supported in part by IST-FET-2005-015905 Mobius project.

† Supported in part by ANR-06-SETIN-010 ParSec project.

‡ Research done while at INRIA Sophia Antipolis - Méditerranée.

al. [13], who adapt CSL from O’Hearn’s simple concurrent language (with a static set of locks and threads) to languages with dynamic lock and thread creation and concurrency primitives that resemble POSIX threads. However, in these variants of CSL, locks are single-entrant; this paper adapts CSL to a Java-like language with *reentrant* locks.

Unfortunately, reentrant locks are inherently problematic for separation-logic reasoning, which tries to completely replace “negative” reasoning about the absence of aliasing by “positive” reasoning about the possession of access permissions. The problem is that a verification system for reentrant locks has to distinguish between initial lock entries and reentries, because only after initial entries is it sound to assume a lock’s resource invariant. This means that initial lock entries need a precondition requiring that the current thread does *not* already hold the acquired lock. Establishing this precondition boils down to proving that the acquired lock does not alias a currently held lock, i.e., to proving absence of aliasing.

This does not mean, however, that permission-based reasoning has to be abandoned altogether for reentrant locks. It merely means that permission-based reasoning alone is insufficient. To illustrate this, we modularly specify and verify a fine-grained lock-coupling list (where lock reentrancy complicates verification) that has previously been verified with separation logic rules for single-entrant locks [10]. This example crucially uses that our verification system includes *value-parameterized types*. Value-parameterized types are generally useful for modularity, and are similar to type-parameterized types in Java Generics [17]. In the lock-coupling example, we use that value-parameterized types can express type-based ownership [7,5], which is a common technique to relieve the aliasing problem in OO verification systems based on classical logic [16].

Another challenge for reasoning about Java-like languages is the handling of inheritance. In Java, each object has an associated reentrant lock, its *object lock*. Naturally, the resource invariant that is associated with an object lock is specified in the object’s class. For subclassing, we need to provide a mechanism for extending resource invariants in subclasses in order to account for extended object state. To this end, we represent resource invariants as abstract predicates [21]. We support modular verification of predicate extensions, by axiomatizing the so-called “stack of class frames” [9,3] in separation logic, as described in our previous work [12].

This paper is structured as follows. First, Section 2 describes the Java-like language that we use for our theoretical development. Next, Section 3 provides some background on separation logic and sketches the axiomatization of the stack of class frames. Section 4 presents Hoare rules for reentrant locking. The rules are illustrated by several examples in Section 5. Last, Section 6 sketches the soundness proof for the verification system, and Section 7 discusses related work and concludes.

## 2 A Java-like Language with Contracts

This section presents the Java-like language that is used to write programs and specifications. The language distinguishes between read-only variables  $l$ , read-write variables  $\ell$ , and logical variables  $\alpha$ . The distinction between read-only and read-write variables is not essential, but often avoids the need for syntactical side conditions in the proof rules (see Section 4 and [12]). Method parameters (including `this`) are read-only; read-write

variables can occur everywhere else, while logical variables can only occur in specifications and types. Apart from this distinction, the *identifier domains* are standard:

$$\begin{aligned} C, D \in \text{ClassId} \quad I \in \text{IntId} \quad s, t \in \text{TypeId} = \text{ClassId} \cup \text{IntId} \quad o, p, q, r \in \text{ObjId} \quad f \in \text{FieldId} \\ m \in \text{MethId} \quad P \in \text{PredId} \quad \iota \in \text{RdVar} \quad \ell \in \text{RdWrVar} \quad \alpha \in \text{LogVar} \\ x, y, z \in \text{Var} = \text{RdVar} \cup \text{RdWrVar} \cup \text{LogVar} \end{aligned}$$

*Values* are integers, booleans, object identifiers and null. For convenience, read-only variables can be used as values directly. Read-only and read-write variables can only contain these basic values, while logical variables range over *specification values* that include both values and *fractional permissions* [6]. Fractional permissions are fractions  $\frac{1}{2^n}$  in the interval  $(0, 1]$ . They are represented symbolically: 1 represents itself, and if symbolic fraction  $\pi$  represents concrete fraction  $fr$  then  $\text{split}(\pi)$  represents  $\frac{1}{2} \cdot fr$ . The full fraction 1 grants *read-write* access right to an associated heap location, while split fractions grant *read-only* access rights. The verification system ensures that the sum of all fractional permissions for the same heap location is always at most 1. As a result, the system prevents read-write and write-write conflicts, while permitting concurrent reads. Formally, the syntactic domain of *values* is defined as follows:

$$\begin{aligned} n \in \text{Int} \quad v, w \in \text{Val} ::= \text{null} \mid n \mid b \mid o \mid \iota \\ b \in \text{Bool} = \{\text{true}, \text{false}\} \quad \pi \in \text{SpecVal} ::= \alpha \mid v \mid 1 \mid \text{split}(\pi) \end{aligned}$$

Now we define the *types* used in our language. Since interfaces and classes (defined next) can be parameterized with specification values, object types are of the form  $t \langle \bar{\pi} \rangle$ . Further, we define special types `perm` (for fractional permissions) and `lockset` (for sets of objects).

$$T, U, V, W \in \text{Type} ::= \text{void} \mid \text{int} \mid \text{bool} \mid t \langle \bar{\pi} \rangle \mid \text{perm} \mid \text{lockset}$$

Next, *class declarations* are defined. Classes declare *fields*, *abstract predicates* (as introduced by Parkinson and Bierman [21]), and *methods*. Following [21], predicates are always implicitly parameterized by the receiver parameter `this`, and can explicitly list additional parameters. Methods have pre/postcondition specifications, parameterized by logical variables. The meaning of a specification is defined via a universal quantification over these parameters. In examples, we usually leave the parameterization implicit, but it is treated explicitly in the formal language.

$$\begin{aligned} F \in \text{Formula} & \quad \text{specification formulas (see Sec. 3 and 4)} \\ \text{spec} ::= \text{req } F; \text{ens } F; & \quad \text{pre/postconditions} \\ \text{fd} ::= T f; & \quad \text{field declarations} \\ \text{pd} ::= \text{pred } P \langle \bar{T} \bar{\alpha} \rangle = F; & \quad \text{predicate definitions} \\ \text{md} ::= \langle \bar{T} \bar{\alpha} \rangle \text{spec } U m(\bar{V} \bar{\iota}) \{c\} & \quad \text{methods (scope of } \bar{\alpha}, \bar{\iota} \text{ is } \bar{T}, \text{spec}, U, \bar{V}, c) \\ \text{cl} \in \text{Class} ::= & \quad \text{classes} \\ \text{class } C \langle \bar{T} \bar{\alpha} \rangle \text{ext } U \text{impl } \bar{V} \{ \text{fd}^* \text{pd}^* \text{md}^* \} & \quad \text{(scope of } \bar{\alpha} \text{ is } \bar{T}, U, \bar{V}, \text{fd}^*, \text{pd}^*, \text{md}^*) \end{aligned}$$

In a similar way, *interfaces* are defined formally as follows:

$$\text{int} \in \text{Interface} ::= \text{interface } I \langle \bar{T} \bar{\alpha} \rangle \text{ext } \bar{U} \{ \text{pt}^* \text{mt}^* \}$$

where  $pt^*$  are predicate types and  $mt^*$  are method types including specifications (see [11] for a formal definition). Class and interface declarations allow to define *class tables*:  $ct \subseteq \text{Interface} \cup \text{Class}$ . We assume that class tables contain the classes `Object` and

Thread. The Thread class declares a `run()` and a `start()` method. The `run()` method is meant to be overridden, whereas the `start()` method is implemented natively and must not be overridden. For thread objects  $o$ , calling  $o.start()$  forks a new thread (whose thread id is  $o$ ) that will execute  $o.run()$ . The `start()`-method has no specification. Instead, our verification system uses `run()`'s precondition as `start()`'s precondition, and `true` as its postcondition.

We impose the following *syntactic restrictions* on interface and class declarations: (1) the types `perm` and `lockset` may only occur inside angle brackets or formulas; (2) cyclic predicate definitions in  $ct$  must be positive. The first restriction ensures that permissions and locksets do not spill into the executable part of the language, while the second ensures that predicate definitions (which can be recursive) are well-founded.

Subtyping, denoted  $<$ , is defined as usual. Commands are sequences of head commands  $hc$  and local variable declarations, terminated by a return value:

$$\begin{aligned} c \in \text{Cmd} &::= v \mid T \ell; c \mid \text{final } T \iota = \ell; c \mid hc; c \\ hc \in \text{HeadCmd} &::= \ell = v \mid \ell = op(\bar{v}) \mid \ell = v.f \mid v.f = v \mid \ell = \text{new } C < \bar{\pi} > \mid \ell = v.m(\bar{v}) \mid \\ &\quad \text{if } (v)\{c\}\text{else}\{c\} \mid v.\text{lock}() \mid v.\text{unlock}() \mid sc \\ sc \in \text{SpecCmd} &::= \text{assert}(F) \mid \pi.\text{commit} \end{aligned}$$

To simplify the proof rules, we assume that programs have been “normalized” prior to verification, so that every intermediate result is assigned to a local variable, and the right hand sides of assignments contain no read-write variables. *Specification commands*  $sc$  are used by the proof system, but are ignored at runtime. The specification command `assert( $F$ )` makes the proof system check that  $F$  holds at this program point, while  `$\pi$ .commit` makes it check that  $\pi$ 's resource invariant is initialized (see Section 4).

### 3 A Variant of Intuitionistic Separation Logic

We now sketch the version of intuitionistic separation logic that we use [12]. *Intuitionistic* separation logic [14,22,21] is suitable for reasoning about properties that are invariant under heap extensions, and is appropriate for garbage-collected languages.

*Specification formulas* are defined by the following grammar:

$$\begin{aligned} lop \in \{*, -*, \&, !\} \quad qt \in \{\text{ex}, \text{fa}\} \quad \kappa \in \text{Pred} &::= P \mid P@C \\ F \in \text{Formula} &::= e \mid \text{PointsTo}(e.f, \pi, e) \mid \pi.\kappa < \bar{\pi} > \mid F \text{lop } F \mid (qt \ T \ \alpha)(F) \end{aligned}$$

We now briefly explain these formulas:

*Expressions*  $e$  are built from values and variables using arithmetic and logical operators, and the operators  $e \text{ instanceof } T$  and  $C \text{ classof } e$ . (The latter holds if  $C$  is  $e$ 's dynamic class.) Expressions of type `bool` are included in the domain of formulas.

The *points-to predicate* `PointsTo( $e.f, \pi, v$ )` is ASCII for  $e.f \xrightarrow{\pi} v$  [4]. Superscript  $\pi$  must be of type `perm` (i.e., a fraction). Points-to has a dual meaning: firstly, it asserts that field  $e.f$  contains value  $v$ , and, secondly, it represents access right  $\pi$  to  $e.f$ . As explained above,  $\pi = 1$  grants write access, and any  $\pi$  grants read access.

The *resource conjunction*  $F * G$  expresses that resources  $F$  and  $G$  are independently available: using either of these resources leaves the other one intact. Resource conjunction is not idempotent:  $F$  does *not* imply  $F * F$ . Because Java is a garbage-collected language, we allow dropping assertions:  $F * G$  implies  $F$ .

The *resource implication*  $F \multimap G$  (a.k.a. *separating implication* or *magic wand*) means “consume  $F$  yielding  $G$ ”. Resource  $F \multimap G$  permits to trade resource  $F$  to receive resource  $G$  in return. Resource conjunction and implication are related by the modus ponens:  $F * (F \multimap G)$  implies  $G$ .

We remark that the logical consequence judgment of our Hoare logic is based on the natural deduction calculus of (*affine*) *linear logic* [23], which coincides with BI’s natural deduction calculus [19] on our restricted set of logical operators. To avoid a proof theory with bunched contexts, we omit the  $\Rightarrow$ -implication between heap formulas (and did not need it in our examples). However, this design decision is not essential.

The *predicate application*  $\pi.\kappa\langle\bar{\pi}\rangle$  applies abstract predicate  $\kappa$  to its receiver parameter  $\pi$  and the additional parameters  $\bar{\pi}$ . As explained above, predicate definitions in classes map abstract predicates to concrete definitions. Predicate definitions can be extended in subclasses to account for extended object state. Semantically,  $P$ ’s predicate extension in class  $C$  gets  $*$ -conjoined with  $P$ ’s predicate extensions in  $C$ ’s superclasses. The *qualified predicate*  $\pi.P@C\langle\bar{\pi}\rangle$  represents the  $*$ -conjunction of  $P$ ’s predicate extensions in  $C$ ’s superclasses, up to and including  $C$ . The *unqualified predicate*  $\pi.P\langle\bar{\pi}\rangle$  is equivalent to  $\pi.P@C\langle\bar{\pi}\rangle$ , where  $C$  is  $\pi$ ’s dynamic class.

The following *derived forms* are convenient:

$$\begin{aligned} \text{PointsTo}(e.f, \pi, T) &\triangleq (\text{ex } T \alpha) (\text{PointsTo}(e.f, \pi, \alpha)) \\ F * \multimap G &\triangleq (F \multimap G) \& (G \multimap F) \quad F \text{ ispartof } G \triangleq G \multimap (F * (F \multimap G)) \end{aligned}$$

Intuitively,  $F \text{ ispartof } G$  says that  $F$  is a physical part of  $G$ : one can take  $G$  apart into  $F$  and its complement  $F \multimap G$ , and can put the two parts together to obtain  $G$  back.

The logical consequence of our Hoare logic is based on the standard natural deduction rules of (*affine*) *linear logic*. Sound *axioms* capture additional properties of our model. We now present some selected axioms<sup>4</sup>:

The following axiom regulates permission accounting ( $\frac{\pi}{2}$  abbreviates  $\text{split}(\pi)$ ):

$$\Gamma \vdash \text{PointsTo}(e.f, \pi, e') * \multimap (\text{PointsTo}(e.f, \frac{\pi}{2}, e') * \text{PointsTo}(e.f, \frac{\pi}{2}, e'))$$

The next axiom allows predicate receivers to toggle between predicate names and predicate definitions. The axiom has the following side conditions:  $\Gamma \vdash \text{this} : C\langle\bar{\pi}''\rangle$ , the extension of  $P\langle\bar{\pi}, \bar{\pi}'\rangle$  in class  $C\langle\bar{\pi}''\rangle$  is  $F$ , and  $C\langle\bar{\pi}''\rangle$ ’s direct supertype is  $D\langle\bar{\pi}\rangle$ :

$$\Gamma \vdash \text{this}.P@C\langle\bar{\pi}, \bar{\pi}'\rangle * \multimap (F * \text{this}.P@D\langle\bar{\pi}\rangle) \quad (\text{Open/Close})$$

Note that  $P@C$  may have more parameters than  $P@D$ : following Parkinson and Bierman [21] we allow subclasses to extend predicate arities. Missing predicate parameters are existentially quantified, as expressed by the following axiom:

$$\Gamma \vdash \pi.P\langle\bar{\pi}\rangle * \multimap (\text{ex } \bar{T} \bar{\alpha}) (\pi.P\langle\bar{\pi}, \bar{\alpha}\rangle) \quad (\text{Missing Parameters})$$

Finally, the following axiom says that a predicate at a receiver’s dynamic type (i.e., without  $@$ -selector) is stronger than the predicate at its static type. In combination with **(Open/Close)**, this allows to open and close predicates at the receiver’s static type:

$$\Gamma \vdash \pi.P@C\langle\bar{\pi}\rangle \text{ ispartof } \pi.P\langle\bar{\pi}\rangle \quad (\text{Dynamic Type})$$

<sup>4</sup> Throughout this paper,  $\Gamma$  ranges over *type environments* assigning types to free variables and object identifiers.

We note that our axioms for abstract predicates formalize the so-called “stack of class frames” [9,3] using separation logic.

Our Hoare rules combine typing judgment with Hoare triples. In a Java-like language, such a combination is needed because method specifications are looked up based on receiver types. As common in separation logic, we use local Hoare rules combined with a frame rule [22]. Except from the rules for reentrant locks, the Hoare rules are pretty standard and we omit them. We point out that we do not admit the structural rule of conjunction. As a result, we do not need to require that resource invariants associated with locks (as presented in Section 4) are precise or supported formulas<sup>5</sup>.

## 4 Proof Rules for Reentrant Locks

We now present the proof rules for reentrant locks: as usual [18], we assign to each lock a *resource invariant*. In our system, resource invariants are distinguished abstract predicates named `inv`. They have a default definition in the `Object` class and are meant to be extended in subclasses:

```
class Object { ... pred inv = true; ... }
```

The resource invariant  $o.\text{inv}$  can be assumed when  $o$ 's lock is acquired non-reentrantly and must be established when  $o$ 's lock is released with its reentrancy level dropping to 0. Regarding the interaction with subclassing, there is nothing special about `inv`. It is treated just like other abstract predicates.

In CSL for single-entrant locks [18], locks can be acquired without precondition. For reentrant locks, on the other hand, it seems unavoidable that the proof rule for acquiring a lock distinguishes between initial acquires and re-acquires. This is needed because it is quite obviously unsound to simply assume the resource invariant after a re-acquire. Thus, a proof system for reentrant locks must keep track of the locks that the current thread holds. To this end, we enrich our specification language:

$$\begin{aligned} \pi \in \text{SpecVal} &::= \dots \mid \text{nil} \mid \pi \cdot \pi \\ F \in \text{Formula} &::= \dots \mid \text{Lockset}(\pi) \mid \pi \text{ contains } e \end{aligned}$$

Here is the informal semantics of the new expressions and formulas:

- `nil`: the empty multiset.
- $\pi \cdot \pi'$ : the multiset union of multisets  $\pi$  and  $\pi'$ .
- `Lockset( $\pi$ )`:  $\pi$  is the multiset of locks held by the current thread. Multiplicities record the current reentrancy level. (*non-copyable*)
- $\pi \text{ contains } e$ : multiset  $\pi$  contains object  $e$ . (*copyable*)

We classify the new formulas (of which there will be two more) into *copyable* and *non-copyable* ones. Copyable formulas represent *persistent state properties* (i.e., properties that hold forever, once established), whereas non-copyable formulas represent *transient state properties* (i.e., properties that hold temporarily). For copyable  $F$ , we postulate the axiom  $(G \ \& \ F) \text{ -* } (G \ * \ F)$ , whereas for non-copyable formulas we postulate no such axiom. Note that this axiom implies  $F \text{ -* } (F \ * \ F)$ , hence the term “copyable”. As indicated above,  $\pi \text{ contains } e$  is copyable, whereas `Lockset( $\pi$ )` is not.

<sup>5</sup> See O’Hearn [18] for definitions of precise and supported formulas, and why they are needed.

*Initial locksets.* When verifying the body of `Thread.run()`, we assume `Lockset(nil)` as a precondition.

*Initializing resource invariants.* Like class invariants must be initialized before method calls, resource invariants must be initialized before the associated locks can be acquired. In O’Hearn’s simple concurrent language [18], the set of locks is static and initialization of resource invariants is achieved in a global initialization phase. This is not possible when locks are created dynamically. Conceivably, we could tie the initialization of resource invariants to the end of object constructors. However, this is problematic because Java’s object constructors are free to leak references to partially constructed objects (e.g., by passing `this` to other methods). Thus, in practice we have to distinguish between initialized and uninitialized objects semantically. Furthermore, a semantic distinction enables late initialization of resource invariants, which can be useful for objects that remain thread-local for some time before getting shared among threads. To support flexible initialization of resource invariants, we introduce two more formulas:

$F \in \text{Formula} ::= \dots \mid e.\text{fresh} \mid e.\text{initialized}$   
*Restriction:*  $e.\text{initialized}$  must not occur in negative positions.

- $e.\text{fresh}$ :  $e$ ’s resource invariant is not yet initialized. (*non-copyable*)
- $e.\text{initialized}$ :  $e$ ’s resource invariant has been initialized. (*copyable*)

The *fresh*-predicate is introduced as a postcondition of `new`:

$$\frac{C \langle \bar{T} \bar{\alpha} \rangle \in ct \quad \Gamma \vdash \bar{\pi} : \bar{T}[\bar{\pi}/\bar{\alpha}] \quad C \langle \bar{\pi} \rangle <: \Gamma(\ell)}{\Gamma \vdash \{\text{true}\} \ell = \text{new } C \langle \bar{\pi} \rangle \{ \ell.\text{init} * C \text{ classof } \ell * \otimes_{\Gamma(u) <: \text{object}} \ell \neq u * \ell.\text{fresh} \}} \text{ (New)}$$

In addition, the postcondition grants access to all fields of the newly created object  $\ell$  (by the special abstract predicate  $\ell.\text{init}$ ), and records that  $\ell$ ’s dynamic class is known to be  $C$ . Furthermore, the postcondition records that the newly created object is distinct from all other objects that are in scope. This postcondition is usually omitted in separation logic, because separation logic gets around explicit reasoning about the absence of aliasing. Unfortunately, we cannot entirely avoid this kind of reasoning when establishing the precondition for the rule **(Lock)** below, which requires that the lock is *not* already held by the current thread.

The specification command  $\pi.\text{commit}$  triggers  $\pi$ ’s transition from the *fresh* to the *initialized* state, provided  $\pi$ ’s resource invariant is established:

$$\frac{\Gamma \vdash \pi : \text{Object} \quad \Gamma \vdash \pi' : \text{lockset}}{\Gamma \vdash \{ \text{Lockset}(\pi') * \pi.\text{inv} * \pi.\text{fresh} \}} \text{ (Commit)}$$

$$\pi.\text{commit}$$

$$\{ \text{Lockset}(\pi') * !(\pi' \text{ contains } \pi) * \pi.\text{initialized} \}$$

*Locking and unlocking.* There are two rules each for locking and unlocking, depending on whether or not the lock/unlock is associated with an initial entry or a reentry:

$$\frac{\Gamma \vdash v : \text{Object} \quad \Gamma \vdash \pi : \text{lockset}}{\Gamma \vdash \{ \text{Lockset}(\pi) * !(\pi \text{ contains } v) * v.\text{initialized} \}} \text{ (Lock)}$$

$$v.\text{lock}()$$

$$\{ \text{Lockset}(v \cdot \pi) * v.\text{inv} \}$$

$$\frac{\Gamma \vdash v : \text{Object} \quad \Gamma \vdash \pi : \text{lockset}}{\Gamma \vdash \{ \text{Lockset}(v \cdot \pi) \} v.\text{lock}() \{ \text{Lockset}(v \cdot v \cdot \pi) \}} \text{ (Re-Lock)}$$

The rule **(Lock)** applies when lock  $v$  is acquired non-reentrantly, as expressed by the precondition  $\text{Lockset}(\pi) * !(\pi \text{ contains } v)$ . The precondition  $v.\text{initialized}$  makes sure that (1) threads only acquire locks whose resource invariant is initialized, and (2) no null-error can happen (because initialized values are non-null). The postcondition adds  $v$  to the current thread's lockset, and assumes  $v$ 's resource invariant. The rule **(Re-Lock)** applies when a lock is acquired reentrantly.

$$\frac{\Gamma \vdash v : \text{Object} \quad \Gamma \vdash \pi : \text{lockset}}{\Gamma \vdash \{\text{Lockset}(v \cdot v \cdot \pi)\} v.\text{unlock}() \{\text{Lockset}(v \cdot \pi)\}} \text{ (Re-Unlock)}$$

$$\frac{\Gamma \vdash v : \text{Object} \quad \Gamma \vdash \pi : \text{lockset}}{\Gamma \vdash \{\text{Lockset}(v \cdot \pi) * v.\text{inv}\} v.\text{unlock}() \{\text{Lockset}(\pi)\}} \text{ (Unlock)}$$

The rule **(Re-Unlock)** applies when  $v$ 's current reentrancy level is at least 2, and **(Unlock)** applies when  $v$ 's resource invariant gets established in the precondition.

*Some non-solutions.* One might wish to avoid the disequalities in **(New)**'s postcondition. Several approaches for this come to mind. First, one could drop the disequalities in **(New)**'s postcondition, and rely on **(Commit)**'s postcondition  $!(\pi' \text{ contains } \pi)$  to establish **(Lock)**'s precondition. While this would be sound, in general it is too weak, as we are not be able to lock  $\pi$  if we first lock some other object  $x$  (because from  $!(\pi' \text{ contains } \pi)$  we cannot derive  $!(x \cdot \pi' \text{ contains } \pi)$  unless we know  $\pi != x$ ). Second, the **Lockset** predicate could be abandoned altogether, using a predicate  $\pi.\text{Held}(n)$  instead, that says that the current thread holds lock  $\pi$  with reentrancy level  $n$ . In particular,  $\pi.\text{Held}(0)$  means that the current thread does not hold  $\pi$ 's lock at all. We could reformulate the rules for locking and unlocking using the **Held**-predicate, and introduce  $\ell.\text{Held}(0)$  as the postcondition of **(New)**, replacing the disequalities. However, this approach does not work, because it grants only the object creator permission to lock the created object! While it is conceivable that a clever program logic could somehow introduce  $\pi.\text{Held}(0)$ -predicates in other ways (besides introducing it in the postcondition of **(New)**), we have not been able to come up with a workable solution along these lines.

## 5 Examples

In this section, we illustrate our proof rules by several examples. We use the following convenient abbreviations:

$$\pi.\text{locked}(\pi') \triangleq \text{Lockset}(\pi \cdot \pi') \quad \pi.\text{unlocked}(\pi') \triangleq \text{Lockset}(\pi') * !(\pi' \text{ contains } \pi)$$

The formula  $\pi.\text{locked}(\pi')$  says that the current thread's lockset  $\pi \cdot \pi'$  contains lock  $\pi$ , and  $\pi.\text{unlocked}(\pi')$  that the current thread's lockset  $\pi'$  does not contain lock  $\pi$ .

*Example 1: A Method with Callee-side Locking.* We begin with a very simple example of a race free implementation of a bank account. The account lock guards access to the account balance, as expressed by `inv`'s definition below.

```
class Account extends Object {
  private int balance;
  pred inv = PointsTo(this.balance, 1, int);
  req this.initialized * this.unlocked(s); ens Lockset(s);
```

```

int deposit(int x) {
  { this.initialized * this.unlocked(s) } (expanding unlocked)
  { this.initialized * Lockset(s) * !(s contains this) }
  lock();
  { Lockset(this.s) * this.inv }
  (opening inv)
  { Lockset(this.s) * PointsTo(this.balance, 1, int) * (this.inv@Account -* this.inv) }
  balance = balance + x;
  { Lockset(this.s) * PointsTo(this.balance, 1, int) * (this.inv@Account -* this.inv) }
  (closing inv)
  { Lockset(this.s) * this.inv }
  unlock();
  { Lockset(s) } } }

```

The precondition of `deposit()` requires that prior to calling `acc.deposit()` the account's resource invariant must be initialized and the current thread must not hold the account lock already. The postcondition ensures that the current thread's lockset after the call equals its lockset before the call. We have annotated `deposit()`'s body with a proof outline and invite the reader to match the outline to our proof rules. Note that when opening `inv`, we use the axioms ([Dynamic Type](#)) and ([Open/Close](#)). When closing `inv`, we use ([Open/Close](#)) and the modus ponens.

*Example 2: A Method with Caller-side Locking.* In the previous example, `deposit()`'s contract does not say that this method updates the account balance. In fact, because our program logic ties the `balance` field to the account's resource invariant, it prohibits the contract to refer to this field unless the account lock is held before and after calling `deposit()`. Note that this is not a shortcoming of our program logic but, on the contrary, is exactly what is needed to ensure sound method contracts: pre/postconditions that refer to the `balance` field when the account object is unlocked are subject to thread interference and thus lead to unsoundness.

However, we can also express a contract for a `deposit()`-method that enforces that callers have acquired the lock prior to calling `deposit()`, and furthermore expresses that `deposit()` updates the `balance` field. To this end, we make use of the feature that the arity of abstract predicates can be extended in subclasses. Thus, we can extend the arity of the `inv`-predicate (which has arity 0 in the `Object` class) to have an additional integer parameter in the `Account` class:

```

class Account extends Object {
  private int balance;
  pred inv<int balance> = PointsTo(this.balance, 1, balance);
  req inv<balance>; ens inv<balance + x>;
  void deposit(int x){ balance = balance + x; } }

```

Here, `deposit()`'s contract is implicitly quantified by the variable `balance`. When a caller establishes the precondition, the `balance` variable gets bound to a concrete integer, namely the current content of the `balance` field. Note that `acc.deposit()` can only be called when `acc` is locked (as locking `acc` is the only way to establish the precondition `acc.inv<_>`). Furthermore, `deposit()`'s contract forces `deposit()`'s implementation to hold the receiver lock on method exit.

*Example 3: A Method Designed for Reentry.* The implementations of the `deposit()` method in the previous examples differ. Because Java’s locks are reentrant, a single implementation of `deposit()` actually satisfies both contracts:

```
class Account extends Object {
  private int balance;
  pred inv<int balance> = PointsTo(this.balance, 1, balance);
  req unlocked(s) * initialized; ens Lockset(s);
  also
  req locked(s) * inv<balance>; ens locked(s) * inv<balance + x>;
  void deposit(int x) { lock(); balance = balance + x; unlock(); } }
```

This example makes use of *contract conjunction*. Intuitively, a method with two contracts joined by “also” satisfies both these contracts. Technically, contract conjunction is a derived form [20]:

$$\begin{aligned} & \text{req } F_1; \text{ens } G_1; \text{also req } F_2; \text{ens } G_2; \\ \stackrel{\Delta}{=} & \text{req } (F_1 \ \& \ \alpha == 1) \mid (F_2 \ \& \ \alpha == 2); \text{ens } (G_1 \ \& \ \alpha == 1) \mid (G_2 \ \& \ \alpha == 2); \end{aligned}$$

In the example, the first clause of the contract conjunction applies when the caller does not yet hold the object lock, and the second clause applies when he already holds it. The precondition `locked(s)` in the second clause is needed as a pre-condition for re-acquiring the lock, see the rule (**Re-Lock**). In Example 2, this precondition was not needed because there `deposit()`’s implementation does not acquire the account lock.

*Example 4: A Fine-grained Locking Policy.* To illustrate that our solution also supports fine-grained locking policies, we show how we can implement lock coupling. Suppose we want to implement a sorted linked list with repetitions. For simplicity, assume that the list has only two methods: `insert()` and `size()`. The former inserts an integer into the list, and the latter returns the current size of the list. To support a constant-time `size()`-method, each node stores the size of its tail in a `count`-field.

In order to allow multiple threads inserting simultaneously, we want to avoid using a single lock for the whole list. We have to be careful, though: a naive locking policy that simply locks one node at a time would be unsafe, because several threads trying to simultaneously insert the same integer can cause a semantic data race, so that some integers get lost and the `count`-fields get out of sync with the list size. The lock coupling technique avoids this by simultaneously holding locks of two neighboring nodes at critical times.

Lock coupling has been used as an example by Gotsman et al. [10] for single-entrant locks. The additional problem with reentrant locks is that `insert()`’s precondition must require that none of the list nodes is in the lockset of the current thread. This is necessary to ensure that on method entry the current thread is capable of acquiring all nodes’s resource invariants:

```
req this.unlocked(s) * no list node is in s; ens Lockset(s);
void insert(int x);
```

The question is how to formally represent the informal condition. Our solution makes use of class parameters. We require that nodes of a lock-coupled list are *statically owned* by the list object, i.e., they have type `Node<o>`, where `o` is the list object. Then we can approximate the above contract as follows:

```

class LockCouplingList implements SortedIntList {
  Node<this> head;
  pred inv<int c> = (ex Node<this> n)(
    PointsTo(head, 1, n) * n.initialized * PointsTo(n.count, 1/2, c) );
  req this.inv<c>; ens this.inv<c> * result==c;
  int size() { return head.count; }
  req Lockset(s) * !(s contains this) * this.traversable(s); ens Lockset(s);
  void insert(int x) {
    lock(); Node<this> n = head;
    if (n!=null) {
      n.lock();
      if (x <= n.val) {
        n.unlock(); head = new Node<this>(x,head); head.commit; unlock();
      } else { unlock(); n.count++; n.insert(x); }
    } else { head = new Node<this>(x,null); unlock(); } } }

class Node<Object owner> implements Owned<owner> {
  int count; int val; Node<owner> next;
  spec_public pred couple<int count.this, int count.next> =
    (ex Node<owner> n)(
      PointsTo(this.count, 1/2, count.this) * PointsTo(this.val, 1, int)
      * PointsTo(this.next, 1, n) * n!=this * n.initialized
      * ( n!=null -* PointsTo(n.count, 1/2, count.next) )
      * ( n==null -* count.this==1 ) );
  spec_public pred inv<int c> = couple<c,c-1>;
  req PointsTo(next.count, 1/2, c);
  ens PointsTo(next.count, 1/2, c)
  * ( next!=null -* PointsTo(this.count, 1, c+1) )
  * ( next==null -* PointsTo(this.count, 1, 1) )
  * PointsTo(this.val, 1, val) * PointsTo(this.next, 1, next);
  Node(int val, Node<owner> next) {
    if (next!=null) { this.count = next.count+1; } else { this.count = 1; }
    this.val = val; this.next = next; }
  req Lockset(this.s) * owner.traversable(s) * this.couple<c+1,c-1>;
  ens Lockset(s);
  void insert(int x) {
    Node<owner> n = next;
    if (n!=null) {
      n.lock();
      if (x <= n.val) {
        n.unlock(); next = new Node<owner>(x,n); next.commit; unlock();
      } else { unlock(); n.count++; n.insert(x); }
    } else { next = new Node<owner>(x, null); unlock(); } } }

```

**Fig. 1.** A lock-coupling list

```

req this.unlocked(s) * no this-owned object is in s; ens Lockset(s);
void insert(int x);

```

To express this formally, we define a marker interface for owned objects:

```

interface Owned<Object owner> { /* a marker interface */ }

```

Next we define an auxiliary predicate  $\pi.traversable(\pi')$  (read as “if the current thread’s lockset is  $\pi'$ , then the aggregate owned by object  $\pi$  is traversable”). Concretely, this predicate says that no object owned by  $\pi$  is contained in  $\pi'$ :

$$\pi.traversable(\pi') \stackrel{\Delta}{=} (\text{fa Object owner, Owned<owner> x})(!(\pi' \text{ contains x}) \mid \text{owner} \neq \pi)$$

Note that in our definition of  $\pi.\text{traversable}(\pi')$ , we quantify over a type parameter (namely the owner-parameter of the `Owned`-type). Here we are taking advantage of the fact that program logic and type system are inter-dependent.

Now, we can formally define an interface for sorted integer lists:

```
interface SortedIntList {
  pred inv<int c>;    // c is the number of list nodes
  req this.inv<c>; ens this.inv<c> * result==c;
  int size();
  req this.unlocked(s) * this.traversable(s); ens Lockset(s);
  void insert(int x); }
```

Figure 1 shows a tail-recursive lock-coupling implementation of `SortedIntList`. It makes use of the predicate modifier `spec_public`, which exports the predicate definition to object clients<sup>6</sup>. The auxiliary predicate  $n.\text{couple}\langle c, c' \rangle$ , as defined in the `Node` class, holds in states where  $n.\text{count} == c$  and  $n.\text{next}.\text{count} == c'$ .

But how can clients of lock-coupling lists establish `insert()`'s precondition? The answer is that client code needs to track the types of locks held by the current thread. For instance, if  $C$  is not a subclass of `Owned`, then `list.insert()`'s precondition is implied by the following assertion, which is satisfied when the current thread has locked only objects of types  $C$  and `Owned<ℓ>`.

```
list.unlocked(s) * ℓ!=list *
(fa Object z)(!(s contains z) | z instanceof C | z instanceof Owned<ℓ>)
```

## 6 Semantics and Soundness

### 6.1 Runtime Structures

We model dynamics by a small-step operational semantics that operates on states, consisting of a heap, a lock table and a thread pool. As usual, *heaps* map each object identifier to its dynamic type and to a mapping from fields to closed values:

$$h \in \text{Heap} = \text{ObjId} \rightarrow \text{Type} \times (\text{FieldId} \rightarrow \text{CVal}) \quad \text{CVal} = \text{Val} \setminus \text{RdVar}$$

*Stacks* map read/write variables to closed values. Their domains do not include read-only variables, because our operational semantics instantiates those by substitution:

$$s \in \text{Stack} = \text{RdWrVar} \rightarrow \text{CVal}$$

A *thread* is a pair of a stack and a command. A *thread pool* maps object identifiers (representing `Thread` objects) to threads. For better readability, we use syntax-like notation and write “ $s$  in  $c$ ” for threads  $t = (s, c)$ , and “ $o_1$  is  $t_1 \mid \dots \mid o_n$  is  $t_n$ ” for thread pools  $ts = \{o_1 \mapsto t_1, \dots, o_n \mapsto t_n\}$ :

$$\begin{aligned} t \in \text{Thread} &= \text{Stack} \times \text{Cmd} && ::= s \text{ in } c \\ ts \in \text{ThreadPool} &= \text{ObjId} \rightarrow \text{Thread} && ::= o_1 \text{ is } t_1 \mid \dots \mid o_n \text{ is } t_n \end{aligned}$$

*Lock tables* map objects  $o$  to either the symbol `free`, or to the thread object that currently holds  $o$ 's lock and a number that counts how often it currently holds this lock:

<sup>6</sup> `spec_public` can be defined in terms of class axioms, see [12].

$$l \in \text{LockTable} = \text{ObjId} \rightarrow \{\text{free}\} \uplus (\text{ObjId} \times \mathbb{N})$$

Finally, a *state* consists of a heap, a lock table, and a thread pool:

$$st \in \text{State} = \text{Heap} \times \text{LockTable} \times \text{ThreadPool}$$

We omit the (pretty standard) rules for our small-step relation  $st \rightarrow_{ct} st'$ . The relation depends on the underlying class table (for looking up methods), hence the subscript  $ct$ .

## 6.2 Kripke Resource Semantics

We define a forcing relation of the form  $\Gamma \vdash \mathcal{E}; \mathcal{R}; s \models F$ , where  $\Gamma$  is a *type environment*,  $\mathcal{E}$  is a *predicate environment*,  $\mathcal{R}$  is a *resource*, and  $s$  is a *stack*. We assume that the stack  $s$ , the formula  $F$ , and the resource  $\mathcal{R}$  are well-typed in  $\Gamma$ , i.e., the semantic relation is defined on well-typed tuples. The predicate environment  $\mathcal{E}$  maps predicate identifiers to concrete heap predicates that satisfy the predicate definitions from the class table. Our well-foundedness restriction on predicate definitions ensures that such a predicate environment exists.

*Resources*  $\mathcal{R}$  range over the set  $\text{Resource}$  with a binary relation  $\# \subseteq \text{Resource} \times \text{Resource}$  (the *compatibility relation*) and a partial binary operator  $* : \# \rightarrow \text{Resource}$  (the *resource joining operator*) that is associative and commutative. Concretely, resources are 5-tuples  $\mathcal{R} = (h, \mathcal{P}, \mathcal{L}, \mathcal{F}, \mathcal{I})$ : a *heap*  $h$ , a *permission table*  $\mathcal{P} \in \text{ObjId} \times \text{FieldId} \rightarrow [0, 1]$ , an *abstract lock table*  $\mathcal{L} \in \text{ObjId} \rightarrow \text{Bag}(\text{ObjId})$ <sup>7</sup>, a *fresh set*  $\mathcal{F} \subseteq \text{ObjId}$ , and an *initialized set*  $\mathcal{I} \subseteq \text{ObjId}$ . We require that resources satisfy the following axioms: (1)  $\mathcal{P}(o, f) > 0$  iff  $o \in \text{dom}(h)$  and  $f \in \text{dom}(h(o)_2)$ , (2)  $\mathcal{F} \cap \mathcal{I} = \emptyset$ , and (3) if  $o \in \mathcal{L}(p)$  then  $o \in \mathcal{I}$ . Each of the five resource components carries itself a resource structure  $(\#, *)$ . These structures are lifted to 5-tuples componentwise. We now define  $\#$  and  $*$  for the five components.

*Heaps* are compatible if they agree on object types and memory content:

$$h \# h' \text{ iff } \left\{ \begin{array}{l} (\forall o \in \text{dom}(h) \cap \text{dom}(h'))( \\ h(o)_1 = h'(o)_1 \text{ and } (\forall f \in \text{dom}(h(o)_2) \cap \text{dom}(h'(o)_2))(h(o)_2(f) = h'(o)_2(f)) \end{array} \right\}$$

To define heap joining, we lift set union to deal with undefinedness:  $f \vee g = f \cup g$ ,  $f \vee \text{undef} = \text{undef} \vee f = f$ . Similarly for types:  $T \vee \text{undef} = \text{undef} \vee T = T \vee T = T$ .

$$(h * h')(o)_1 \triangleq h(o)_1 \vee h'(o)_1 \quad (h * h')(o)_2 \triangleq h(o)_2 \vee h'(o)_2$$

Joining *permission tables* is pointwise addition:

$$\mathcal{P} \# \mathcal{P}' \text{ iff } (\forall o)(\mathcal{P}(o) + \mathcal{P}'(o) \leq 1) \quad (\mathcal{P} * \mathcal{P}')(o) \triangleq \mathcal{P}(o) + \mathcal{P}'(o)$$

*Abstract lock tables* map thread identifiers to locksets. The compatibility relation captures that distinct threads cannot hold the same lock.

$$\mathcal{L} \# \mathcal{L}' \text{ iff } \left\{ \begin{array}{l} \text{dom}(\mathcal{L}) \cap \text{dom}(\mathcal{L}') = \emptyset \\ (\forall o \in \text{dom}(\mathcal{L}), p \in \text{dom}(\mathcal{L}'))(\mathcal{L}(o) \sqcap \mathcal{L}'(p) = []) \end{array} \right. \quad \mathcal{L} * \mathcal{L}' \triangleq \mathcal{L} \sqcup \mathcal{L}'$$

*Fresh sets*  $\mathcal{F}$  keep track of allocated but not yet initialized objects, while *initialized sets*  $\mathcal{I}$  keep track of initialized objects. We define  $\#$  for fresh sets as disjointness in order to mirror that  $o.\text{fresh}$  is non-copyable, and for initialized sets as equality in order to mirror that  $o.\text{initialized}$  is copyable:

<sup>7</sup> Where we use  $\sqcap$  to denote bag intersection,  $\sqcup$  for bag union, and  $[]$  for the empty bag.

$$\begin{aligned}
\mathcal{F} \# \mathcal{F}' &\text{ iff } \mathcal{F} \cap \mathcal{F}' = \emptyset & \mathcal{F} * \mathcal{F}' &\triangleq \mathcal{F} \cup \mathcal{F}' \\
\mathcal{I} \# \mathcal{I}' &\text{ iff } \mathcal{I} = \mathcal{I}' & \mathcal{I} * \mathcal{I}' &\triangleq \mathcal{I} (= \mathcal{I}')
\end{aligned}$$

This completes the description of the semantic domains. We continue with the formal semantics of expressions and formulas. Expressions of type `lockset` are interpreted as multisets in the obvious way:  $\llbracket \text{nil} \rrbracket_s^h = []$  and  $\llbracket e \cdot e' \rrbracket_s^h = \llbracket e \rrbracket_s^h \sqcup \llbracket e' \rrbracket_s^h$ . Here are the semantic clauses for our new formulas for reentrant locking:

$$\begin{aligned}
\Gamma \vdash \mathcal{E}; (h, \mathcal{P}, \mathcal{L}, \mathcal{F}, \mathcal{I}); s \models \text{Lockset}(\pi) &\text{ iff } \mathcal{L}(o) = \llbracket \pi \rrbracket \text{ for some } o \\
\Gamma \vdash \mathcal{E}; (h, \mathcal{P}, \mathcal{L}, \mathcal{F}, \mathcal{I}); s \models \pi \text{ contains } e &\text{ iff } \llbracket e \rrbracket_s^h \in \llbracket \pi \rrbracket \\
\Gamma \vdash \mathcal{E}; (h, \mathcal{P}, \mathcal{L}, \mathcal{F}, \mathcal{I}); s \models e.\text{fresh} &\text{ iff } \llbracket e \rrbracket_s^h \in \mathcal{F} \\
\Gamma \vdash \mathcal{E}; (h, \mathcal{P}, \mathcal{L}, \mathcal{F}, \mathcal{I}); s \models e.\text{initialized} &\text{ iff } \llbracket e \rrbracket_s^h \in \mathcal{I}
\end{aligned}$$

These clauses are self-explanatory, except perhaps the existential quantification in the clause for `Lockset`( $\pi$ ). Intuitively, this clause says that there exists a thread identifier  $o$  in the domain of  $\mathcal{L}$  such that  $\pi$  denotes the current lockset associated with  $o$ . We omit the (standard) clauses for the other logical operators, see e.g., [11].

### 6.3 Soundness

In this section, we extend our verification rules to runtime states. The extended rules are never used in verification, but instead define a global state invariant,  $st : \diamond$ , that is preserved by the small-step rules of our operational semantics.

We need a few definitions: For  $\mathcal{R} = (h, \mathcal{P}, \mathcal{L}, \mathcal{F}, \mathcal{I})$ , let  $\mathcal{R}_{\text{hp}} = h$ ,  $\mathcal{R}_{\text{perm}} = \mathcal{P}$ ,  $\mathcal{R}_{\text{lock}} = \mathcal{L}$ ,  $\mathcal{R}_{\text{fresh}} = \mathcal{F}$  and  $\mathcal{R}_{\text{init}} = \mathcal{I}$ . Our forcing relation  $\models$  from the last section assumes formulas without logical variables: we deal with those by substitution, ranged over by  $\sigma \in \text{LogVar} \rightarrow \text{SpecVal}$ . We state  $(\Gamma \vdash \sigma : \Gamma')$  whenever  $\text{dom}(\sigma) = \text{dom}(\Gamma')$  and  $(\Gamma[\sigma] \vdash \sigma(\alpha) : \Gamma'(\alpha)[\sigma])$  for all  $\alpha$  in  $\text{dom}(\sigma)$ . Furthermore, we define  $\text{cfv}(c) = \{x \in \text{fv}(c) \mid x \text{ occurs in an object creation command } \ell = \text{new } C \langle \bar{\pi} \rangle\}$ .

Now, we extend the Hoare triple judgment to threads:

$$\frac{\Gamma_{\text{hp}} = \text{fst} \circ \mathcal{R}_{\text{hp}} \quad \Gamma \vdash \sigma : \Gamma' \quad \text{dom}(\Gamma') \cap \text{cfv}(c) = \emptyset \quad \Gamma, \Gamma' \vdash s : \diamond \quad \text{dom}(\mathcal{R}_{\text{lock}}) \subseteq \{o\} \quad \Gamma[\sigma] \vdash \mathcal{E}; \mathcal{R}; s \models F[\sigma] \quad \Gamma, \Gamma'; r \vdash \{F\}c : \text{void}\{G\}}{\mathcal{R} \vdash o \text{ is } (s \text{ in } c) : \diamond} \text{ (Thread)}$$

The object identifier  $r$  in the Hoare triple (last premise) is the current receiver, needed to determine the scope of abstract predicates. We have omitted the receiver parameter from our Hoare rules in Section 4, because for source code verification the receiver parameter is always `this`.

We straightforwardly extend this judgment to thread pools:

$$\frac{}{\mathcal{R} \vdash \emptyset : \diamond} \text{ (Empty Pool)} \quad \frac{\mathcal{R} \vdash t : \diamond \quad \mathcal{R}' \vdash ts : \diamond}{\mathcal{R} * \mathcal{R}' \vdash t \mid ts : \diamond} \text{ (Cons Pool)}$$

To further extend the judgment to states, we define the set  $\text{ready}(\mathcal{R})$  of all initialized objects whose locks are not held, and the function  $\text{conc}$  that maps abstract lock tables to concrete lock tables:

$$\begin{aligned}
\text{ready}(\mathcal{R}) &\triangleq \mathcal{R}_{\text{init}} \setminus \{o \mid (\exists p)(o \in \mathcal{L}(p))\} \\
\text{conc}(\mathcal{L})(o) &\triangleq (p, \mathcal{L}(p)(o)), \text{ if } o \in \mathcal{L}(p) \quad \text{conc}(\mathcal{L})(o) \triangleq \text{free}, \text{ otherwise}
\end{aligned}$$

In `conc`'s definition, we let  $\mathcal{L}(p)(o)$  stand for the multiplicity of  $o$  in  $\mathcal{L}(p)$ . Note that `conc` is well-defined, by axiom (2) for resources. The rule for states ensures that there exists a resource  $\mathcal{R}$  to satisfy the thread pool  $ts$ , and a resource  $\mathcal{R}'$  to satisfy the resource invariants of the locks that are ready to be acquired:

$$\frac{\mathcal{R}\#\mathcal{R}' \quad \mathcal{R}'_{\text{lock}} = \mathbf{0} \quad \text{fst} \circ \mathcal{R}'_{\text{hp}} \subseteq \text{fst} \circ h = \Gamma \quad \Gamma \vdash \mathcal{E}; \mathcal{R}'; \mathbf{0} \models \otimes_{o \in \text{ready}(\mathcal{R})} o.\text{inv}}{h = (\mathcal{R} * \mathcal{R}')_{\text{hp}} \quad l = \text{conc}(\mathcal{R}_{\text{lock}}) \quad \mathcal{R} \vdash ts : \diamond} \langle h, l, ts \rangle : \diamond \quad (\text{State})$$

The judgment  $(ct : \diamond)$  is the top-level judgment of our source code verification system, to be read as “class table  $ct$  is verified”. We have shown the following theorem:

**Theorem 1 (Preservation).** *If  $(ct : \diamond)$ ,  $(st : \diamond)$  and  $st \rightarrow_{ct} st'$ , then  $(st' : \diamond)$ .*

From the preservation theorem, we can draw the following corollaries: verified programs are data race free, verified programs never dereference `null`, and if a verified program contains `assert(F)`, then  $F$  holds whenever the assertion is reached.

## 7 Comparison to Related Work and Conclusion

*Related work.* There are a number of similarities between our work and Gotsman et al. [10], for instance the treatment of initialization of dynamically created locks. Our `initialized` predicate corresponds to what Gotsman calls lock handles (with his lock handle parameters corresponding to our class parameters). Since Gotsman’s language supports deallocation of locks, he scales lock handles by fractional permissions in order to keep track of sharing. This is not necessary in a garbage-collected language. In addition to single-entrant locks, Gotsman also treats thread joining. We have covered joining in a recent paper [12] for Java threads (joining Java threads has a slightly different operational semantics than joining POSIX threads as modeled in [10]). The essential differences between Gotsman’s and our paper are (1) that we treat reentrant locks, which are a different synchronization primitive than single-entrant locks, and (2) that we treat subclassing and extension of resource invariants in subclasses. Hobor et al.’s work [13] is very similar to [10].

Another related line of work is by Jacobs et al. [15] who extend the Boogie methodology for reasoning about object invariants [3] to a multithreaded Java-like language. While their system is based on classical logic (without operators like `*` and `-*`), it includes built-in notions of ownership and access control. Their system deliberately enforces a certain programming discipline (like CSL and our variant of it also do) rather than aiming for a complete program logic. The object life cycle imposed by their discipline is essentially identical to ours. For instance, their shared objects (objects that are shared between threads) directly correspond to our `initialized` objects (objects whose resource invariants are initialized). Their system prevents deadlocks, which our system does not. They achieve deadlock prevention by imposing a partial order on locks. As a consequence of their order-based deadlock prevention, their programming discipline statically prevents reentrancy, although it may not be too hard to relax this at the cost of additional complexity.

In a more traditional approach, Ábráham, De Boer et al. [1,8] apply assume-guarantee reasoning to a multithreaded Java-like language.

*Conclusion.* We have adapted concurrent separation logic to a Java-like language. Resource invariants are specified as abstract predicates in classes, and can be modularly extended in subclasses by a separation-logic axiomatization of the “stack of class frames” [9,3]. The main difficulty was dealing with reentrant locks. These complicate the proof rules, and some reasoning about the absence of aliasing is needed. However, permission-based reasoning is still largely applicable, as illustrated by a verification of a lock-coupling list in spite of reentrancy. In this example, a rich dependent type system with value-parameterized classes proved useful. Because we needed to extend CSL’s proof rules to support reasoning about the absence of aliasing (e.g., by adding an additional postcondition to the object creation rule), it does not seem possible to derive our proof rules from CSL’s standard proof rules through an encoding of reentrant locks in terms of single-entrant locks. We have omitted `wait/notify` (conditional synchronization) in this paper, but we have treated it in our technical report [11]. Whereas reentrancy slightly complicates the operational semantics of `wait/notify` (because the runtime has to remember the reentrancy level of a waiting thread), the proof rules for `wait/notify` are unproblematic.

## References

1. E. Ábrahám, F. S. de Boer, W.-P. de Roever, M. Steffen. Tool-supported proof system for multithreaded Java. In F. S. de Boer, M. M. Bonsangue, S. Graf, W.-P. de Roever, eds., *Formal Methods for Components and Objects*, vol. 2852 of *Lecture Notes in Computer Science*. Springer-Verlag, 2003.
2. G. Andrews. *Concurrent Programming: Principles and Practice*. Benjamin/Cummings, 1991.
3. M. Barnett, R. DeLine, M. Fähndrich, K. R. M. Leino, W. Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3(6), 2004.
4. R. Bornat, P. W. O’Hearn, C. Calcagno, M. Parkinson. Permission accounting in separation logic. In *Principles of Programming Languages*, New York, NY, USA, 2005. ACM Press.
5. C. Boyapati, R. Lee, M. Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, 2002.
6. J. Boyland. Checking interference with fractional permissions. In R. Cousot, ed., *Static Analysis Symposium*, vol. 2694 of *Lecture Notes in Computer Science*. Springer-Verlag, 2003.
7. D. G. Clarke, J. M. Potter, J. Noble. Ownership types for flexible alias protection. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, vol. 33:10 of *ACM SIGPLAN Notices*, New York, 1998. ACM Press.
8. F. S. de Boer. A sound and complete shared-variable concurrency model for multi-threaded Java programs. In *International Conference on Formal Methods for Open Object-based Distributed Systems*, 2007.
9. R. DeLine, M. Fähndrich. Typestates for objects. In *European Conference on Object-Oriented Programming*, 2004.
10. A. Gotsman, J. Berdine, B. Cook, N. Rinetzky, M. Sagiv. Local reasoning for storable locks and threads. In *Asian Programming Languages and Systems Symposium*, 2007.
11. C. Haack, M. Huisman, C. Hurlin. Reasoning about Java’s reentrant locks. Technical Report ICIS-R08014, Radboud University Nijmegen, 2008.
12. C. Haack, C. Hurlin. Separation logic contracts for a Java-like language with `fork/join`. In *Algebraic Methodology and Software Technology*, number 5140 in *Lecture Notes in Computer Science*. Springer-Verlag, 2008.
13. A. Hobor, A. Appel, F. Nardelli. Oracle semantics for concurrent separation logic. In *European Symposium on Programming*, vol. 4960 of *Lecture Notes in Computer Science*, 2008.
14. S. Ishtiaq, P. W. O’Hearn. BI as an assertion language for mutable data structures. In *Principles of Programming Languages*, 2001.
15. B. Jacobs, J. Smans, F. Piessens, W. Schulte. A statically verifiable programming model for concurrent object-oriented programs. In *International Conference on Formal Engineering Methods*, 2006.
16. P. Müller. *Modular Specification and Verification of Object-Oriented Programs*, vol. 2262 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.
17. M. Naftalin, P. Wadler. *Java Generics*. O’Reilly, 2006.
18. P. W. O’Hearn. Resources, concurrency and local reasoning. *Theoretical Computer Science*, 375(1–3), 2007.
19. P. W. O’Hearn, D. J. Pym. The logic of bunched implications. *Bulletin of Symbolic Logic*, 5(2), 1999.
20. M. Parkinson. *Local Reasoning for Java*. PhD thesis, University of Cambridge, 2005.
21. M. Parkinson, G. Bierman. Separation logic and abstraction. In *Principles of Programming Languages*, 2005.
22. J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Logic in Computer Science*, Copenhagen, Denmark, 2002. IEEE Press.
23. P. Wadler. A taste of linear logic. In *Mathematical Foundations of Computer Science*, 1993.