Reasoning about Java's Reentrant Locks

Christian Haack[1⋆], Marieke Huisman[2⋆†] and Clément Hurlin[2⋆†]

[1] Radboud Universiteit Nijmegen, The Netherlands
[2] 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-parametrized 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.

⋆ Supported in part by IST-FET-2005-015905 Mobius project.
† Supported in part by ANR-06-SETIN-010 ParSec project.

Table of Contents

1 Overview and Examples 3
- 1 Introduction 3
- 2 A Java-like Language with Contracts 4
- 3 A Variant of Intuitionistic Separation Logic 6
- 4 Proof Rules for Reentrant Locks 7
- 5 Examples 10
- 6 Semantics and Soundness 14
  - 6.1 Runtime Structures 14
  - 6.2 Kripke Resource Semantics 15
  - 6.3 Soundness 16
- 7 Comparison to Related Work and Conclusion 17

2 Technical Appendix 20
- A Notational Conventions 20
- B Class Tables, Interfaces, Subtyping (Appendix for Section 2) 20
- C Auxiliary Functions 22
- D Type Environments and Types 24
- E Values and Specification Values 24
- F Expressions 25
- G Formulas 27
- H Objects, Heaps, Stacks 27
- I Proof Theory 28
  - I.1 Natural Deduction Rules 28
  - I.2 Axioms 29
- J Hoare Triples 30
- K Verified Interfaces and Classes 31
- L Operational Semantics 33
- M Predicate Environments 36
- N Kripke Resource Semantics 37
- O Properties 38
  - O.1 Properties of the Typing Judgments 39
  - O.2 Properties of the Proof Theory 40
  - O.3 Properties of Hoare Triples 41
  - O.4 Properties of the Semantics 42
- P Preservation 43
- Q Data Race Freedom, Error Freedom and Partial Correctness 50
  - Q.1 Data Race Freedom 51
  - Q.2 Null Error Freedom 51
  - Q.3 No Illegal Monitor States 52
  - Q.4 Partial Correctness 52

1

Overview and Examples

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) [19]. 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 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 (in spite of reentrant locks) that has previously been verified with separation logic rules for single-entrant locks [10]. In this example, we crucially use that our verification system includes *value-parametrized types*. Value-parametrized types

are generally useful for modularity, and are similar to type-parameterized types in Java Generics [18]. In the lock-coupling example, we use that value-parametrized 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 [17].

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 [22]. 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 [11].

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 ι, read-write variables ℓ, and logical variables α. 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 [11]). 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:

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

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 π represents concrete fraction fr then $\mathtt{split}(\pi)$ 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 right. 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:

$$n \in \mathsf{Int} \qquad u,v,w \in \mathsf{Val} ::= \mathtt{null} \mid n \mid b \mid o \mid \iota$$
$$b \in \mathsf{Bool} = \{\mathtt{true},\mathtt{false}\} \qquad \pi \in \mathsf{SpecVal} ::= \alpha \mid v \mid 1 \mid \mathtt{split}(\pi)$$

Now we define the types used in our language. First, notice that since interfaces and classes (defined next) can be parametrized with specification values, object types are of the form $t\texttt{<}\bar{\pi}\texttt{>}$. Further, we define special types `perm` (for fractional permissions) and `lockset` (for sets of objects).

$$T,U,V,W \in \mathsf{Type} ::= \texttt{void} \mid \texttt{int} \mid \texttt{bool} \mid t\texttt{<}\bar{\pi}\texttt{>} \mid \texttt{perm} \mid \texttt{lockset}$$

Next, *class declarations* are defined. Classes declare *fields*, *abstract predicates* (as introduced by Parkinson and Bierman [22]), and *methods*. Optionally, class-, method- and predicate declarations can be preceded by a `final`-modifier to prohibit subclassing (for classes), overriding (for methods) and extension (for predicates). Following [22], abstract predicates are always implicitly parametrized by the receiver parameter `this`, and can explicitly list additional parameters. Methods have pre/postcondition specifications parametrized by logical variables. The meaning of a specification is defined via a universal quantification over these parameters. In examples, we usually leave the parametrization implicit, but it is treated explicitly in the formal language.

$fin ::= \texttt{final?}$	optional modifier
$F \in \mathsf{Formula}$	specification formulas (see Sec. 3 and 4)
$spec ::= \texttt{req}\,F\texttt{;}\,\texttt{ens}\,F\texttt{;}$	pre/postconditions
$fd ::= T\ f\texttt{;}$	field declarations
$pd ::= fin\ \texttt{pred}\ P\texttt{<}\bar{T}\ \bar{\alpha}\texttt{>=}F\texttt{;}$	predicate definitions
$md ::= fin\ \texttt{<}\bar{T}\ \bar{\alpha}\texttt{>}spec\ U\ m\texttt{(}\bar{V}\,\bar{\imath}\texttt{)\{}c\texttt{\}}$	methods (scope of $\bar{\alpha},\bar{\imath}$ is $\bar{T},spec,U,\bar{V},c$)
$cl \in \mathsf{Class} ::=$	classes
$fin\ \texttt{class}\ C\texttt{<}\bar{T}\ \bar{\alpha}\texttt{>}\ \texttt{ext}\ U\ \texttt{impl}\ \bar{V}\ \texttt{\{}fd\texttt{*}\ pd\texttt{*}\ md\texttt{*\}}$	(scope of $\bar{\alpha}$ is $\bar{T},U,\bar{V},fd\texttt{*},pd\texttt{*},md\texttt{*}$)

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

$$int \in \mathsf{Interface} ::= \texttt{interface}\ I\texttt{<}\bar{T}\ \bar{\alpha}\texttt{>}\ \texttt{ext}\ \bar{U}\ \texttt{\{}pt\texttt{*}\ mt\texttt{*\}}$$

where pt* are predicate types and mt* are method types including specifications (see Appendix B for a formal definition). Class and interface declarations allow to define *class tables*: $ct \subseteq \mathsf{Interface} \cup \mathsf{Class}$. 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 must not be overridden and is implemented natively. For thread objects o, calling $o\texttt{.start()}$ forks a new thread (whose thread id is o) and executes $o\texttt{.run()}$ in this thread. 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.

The symbol $<:$ denotes subtyping and is defined as usual (see Appendix B for details). Commands are sequences of head commands hc and local variable declarations, terminated by a return value:

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

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 π`.commit` makes it check that π's resource invariant is initialized (see Section 4).

3 A Variant of Intuitionistic Separation Logic

We now sketch the version [11] of intuitionistic separation logic [15,23,22] that we use. *Intuitionistic* separation logic 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:

$$lop \in \{\ast, \texttt{-*}, \texttt{\&}, \texttt{|}\} \qquad qt \in \{\texttt{ex}, \texttt{fa}\} \qquad \kappa \in \mathsf{Pred} ::= P \mid P\texttt{@}C$$
$$F \in \mathsf{Formula} ::= e \mid \texttt{PointsTo(}e.f, \pi, e\texttt{)} \mid \pi.\kappa\texttt{<}\bar{\pi}\texttt{>} \mid F\ lop\ F \mid \texttt{(}qt\ T\ \alpha\texttt{)(}F\texttt{)}$$

We now briefly explain these formulas:

Expressions e are built from values and variables using arithmetic and logical operators, and the operators e `instanceof` T and C `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 π 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 π to $e.f$. As explained above, $\pi = 1$ grants write access, and any π 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 `-*` G (a.k.a. *separating implication* or *magic wand*) means "consume F yielding G". Resource F `-*` G permits to trade resource F to receive resource G in return. Resource conjunction and implication are related by the modus ponens: $F * (F$ `-*` $G)$ implies G.

We remark that the logical consequence judgment of our Hoare logic is based on the natural deduction calculus of *(affine) linear logic* [24], which coincides with BI's natural deduction calculus [20] 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* $\pi.\kappa\texttt{<}\bar{\pi}\texttt{>}$ applies abstract predicate κ to its receiver parameter π 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* $\pi.P\texttt{@}C\texttt{<}\bar{\pi}\texttt{>}$ represents the $*$-conjunction of P's predicate extensions in C's superclasses, up to and including C. The *unqualified predicate* $\pi.P\texttt{<}\bar{\pi}\texttt{>}$ is equivalent to $\pi.P\texttt{@}C\texttt{<}\bar{\pi}\texttt{>}$, where C is π's dynamic class.

The following *derived forms* are convenient:

$$\texttt{PointsTo}(e.f,\pi,T) \triangleq (\texttt{ex}\ T\ \alpha)(\ \texttt{PointsTo}(e.f,\pi,\alpha))$$
$$F \mathbin{*\!\!-\!\!*} G \triangleq (F \mathbin{-\!\!*} G)\ \&\ (G \mathbin{-\!\!*} F) \qquad F\ \texttt{ispartof}\ G \triangleq G \mathbin{-\!\!*} (F * (F \mathbin{-\!\!*} G))$$

Intuitively, F `ispartof` G says that F is a physical part of G: one can take G apart into F and its complement $F \mathbin{-\!\!*} G$, 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[3]:

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

$$\Gamma \vdash \texttt{PointsTo}(e.f,\pi,e') \mathbin{*\!\!-\!\!*} (\texttt{PointsTo}(e.f,\tfrac{\pi}{2},e') * \texttt{PointsTo}(e.f,\tfrac{\pi}{2},e'))$$

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

$$\Gamma \vdash \texttt{this}.P\texttt{@}C\texttt{<}\bar{\pi},\bar{\pi}'\texttt{>} \mathbin{*\!\!-\!\!*} (F * \texttt{this}.P\texttt{@}D\texttt{<}\bar{\pi}\texttt{>}) \qquad \text{(Open/Close)}$$

Note that $P\texttt{@}C$ may have more parameters than $P\texttt{@}D$: following Parkinson and Bierman [22] we allow subclasses to extend predicate arities. Missing predicate parameters are existentially quantified, as expressed by the following axiom:

$$\Gamma \vdash \pi.P\texttt{<}\bar{\pi}\texttt{>} \mathbin{*\!\!-\!\!*} (\texttt{ex}\ \bar{T}\ \bar{\alpha})(\pi.P\texttt{<}\bar{\pi},\bar{\alpha}\texttt{>}) \qquad \text{(Missing Parameters)}$$

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:

$$\Gamma \vdash \pi.P\texttt{@}C\texttt{<}\bar{\pi}\texttt{>}\ \texttt{ispartof}\ \pi.P\texttt{<}\bar{\pi}\texttt{>} \qquad \text{(Dynamic Type)}$$

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 [23]. 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[4].

4 Proof Rules for Reentrant Locks

We now present the proof rules for reentrant locks: as usual [19], 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; ... }
```

[3] Throughout this paper, Γ ranges over *type environments* assigning types to free variables and object identifiers.

[4] See O'Hearn [19] for definitions of precise and supported formulas, and why they are needed.

The resource invariant o.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 [19], 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 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:

$$\begin{array}{l} \pi \in \mathsf{SpecVal} \ ::= \ \ldots \mid \texttt{nil} \mid \pi \cdot \pi \\ F \in \mathsf{Formula} \ ::= \ \ldots \mid \texttt{Lockset}(\pi) \mid \pi \ \texttt{contains} \ e \end{array}$$

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

- nil: the empty multiset.
- $\pi \cdot \pi'$: the multiset union of multisets π and π'.
- Lockset(π): π is the multiset of locks held by the current thread. Multiplicities record the current reentrancy level. *(non-copyable)*
- π contains e: multiset π 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 $(G \,\&\, F) \mathrel{-\!\!*} (G * F)$, whereas for non-copyable formulas we postulate no such axiom. Note that this axiom implies $F \mathrel{-\!\!*} (F * F)$, hence the term "copyable". As indicated above, π contains e is copyable, whereas Lockset(π) is not.

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 [19], 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:

$$\begin{array}{l} F \in \mathsf{Formula} \ ::= \ \ldots \mid e.\texttt{fresh} \mid e.\texttt{initialized} \\ \textit{Restriction: } e.\texttt{initialized} \text{ must not occur in negative positions.} \end{array}$$

- e.fresh: e's resource invariant is not yet initialized. *(non-copyable)*
- e.initialized: e's resource invariant has been initialized. *(copyable)*

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

$$\frac{C\texttt{<}\bar{T}\ \bar{\alpha}\texttt{>} \in ct \quad \Gamma \vdash \bar{\pi} : \bar{T}[\bar{\pi}/\alpha] \quad C\texttt{<}\bar{\pi}\texttt{>} <: \Gamma(\ell)}{\Gamma \vdash \{\texttt{true}\}\ell\texttt{=new}\ C\texttt{<}\bar{\pi}\texttt{>}\{\ell\texttt{.init} * C\ \texttt{classof}\ \ell * \circledast_{\Gamma(u)<:\texttt{Object}}\ \ell\ \texttt{!=}\ u * \ell\texttt{.fresh}\}} \text{ (New)}$$

In addition, the postcondition grants access to all fields of the newly created object ℓ (by the special abstract predicate ℓ`.init`), and records that ℓ'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 π`.commit` triggers π's transition from the `fresh` to the `initialized` state, provided π's resource invariant is established:

$$\frac{\Gamma \vdash \pi : \texttt{Object} \quad \Gamma \vdash \pi' : \texttt{lockset}}{\begin{array}{c}\Gamma \vdash \{\texttt{Lockset(}\pi'\texttt{)} * \pi\texttt{.inv} * \pi\texttt{.fresh}\} \\ \pi\texttt{.commit} \\ \{\texttt{Lockset(}\pi'\texttt{)} * \texttt{!(}\pi'\ \texttt{contains}\ \pi\texttt{)} * \pi\texttt{.initialized}\}\end{array}} \text{ (Commit)}$$

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:

$$\frac{\Gamma \vdash v : \texttt{Object} \quad \Gamma \vdash \pi : \texttt{lockset}}{\begin{array}{c}\Gamma \vdash \{\texttt{Lockset(}\pi\texttt{)} * \texttt{!(}\pi\ \texttt{contains}\ v\texttt{)} * v\texttt{.initialized}\} \\ v\texttt{.lock()} \\ \{\texttt{Lockset(}v \cdot \pi\texttt{)} * v\texttt{.inv}\}\end{array}} \text{ (Lock)}$$

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

The rule (Lock) applies when lock v is acquired non-reentrantly, as expressed by the precondition `Lockset(`π`) * !(`π `contains` v`)`. The precondition v`.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.

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

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

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. We discuss some ideas that try to avoid the disequalities in (New)'s postcondition. The first idea drops the disequalities in (New)'s postcondition, and relies on (Commit)'s postcondition $\texttt{!}(\pi' \texttt{ contains } \pi)$ for establishing (Lock)'s precondition. While this would be sound, it is too weak in general, as we would not be able to lock π if we first locked some other object x (because from $\texttt{!}(\pi' \texttt{ contains } \pi)$ we cannot derive $\texttt{!}(x \cdot \pi' \texttt{ contains } \pi)$ unless we know $\pi \texttt{ != } x$). The second idea abandons the `Lockset` predicate altogether, and instead uses a predicate $\pi\texttt{.Held}(n)$ that says that the current thread holds lock π with reentrancy level n. In particular, $\pi\texttt{.Held(0)}$ means that the current thread does not hold π's lock at all. We could reformulate the rules for locking and unlocking using the `Held`-predicate, and introduce $\ell\texttt{.Held(0)}$ as the postcondition of (New), replacing the disequalities. Unfortunately, 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 $\pi\texttt{.Held(0)}$-predicates in other ways (in addition to as a postcondition of (New)), we were unable 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:

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

The formula $\pi\texttt{.locked}(\pi')$ says that lock π is contained in the current thread's lockset π', and $\pi\texttt{.unlocked}(\pi')$ that π is not contained in the current thread's lockset π'.

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.

Fortunately, we can 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 [21]:

```
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

$$\begin{array}{rl} & \texttt{req}\ F_1; \texttt{ens}\ G_1;\ \texttt{also req}\ F_2; \texttt{ens}\ G_2; \\ \triangleq & \texttt{req}\ (F_1\ \&\ \alpha == 1) \mid (F_2\ \&\ \alpha == 2);\ \texttt{ens}\ (G_1\ \&\ \alpha == 1) \mid (G_2\ \&\ \alpha == 2); \end{array}$$

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 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 represent the informal condition in italic. 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:

```
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 e`.traversable<`s`>` (read as "if the current thread's lockset is s, then the aggregate owned by object e is traversable"). Concretely, this predicate says that no object owned by e is contained in s:

$$e\texttt{.traversable<}e'\texttt{>} \triangleq \texttt{(fa Object owner, Owned<owner> x)(!(}e'\texttt{ contains x) | owner !=}e\texttt{)}$$

Note that in our definition of e`.traversable<`e'`>`, we quantify over a type parameter (namely the `owner`-parameter of the `Owned`-type). We are here 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[5]. The auxiliary predicate n.`couple<`c,c'`>`, as defined in the `Node` class, holds in states where n.`count ==` c and n.`next.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 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:

$$h \in \mathsf{Heap} = \mathsf{ObjId} \rightharpoonup \mathsf{Type} \times (\mathsf{FieldId} \rightharpoonup \mathsf{ClVal}) \qquad \mathsf{ClVal} = \mathsf{Val} \setminus \mathsf{RdVar}$$

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

$$s \in \mathsf{Stack} = \mathsf{RdWrVar} \rightharpoonup \mathsf{ClVal}$$

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 $t_1 \mid \cdots \mid o_n$ is t_n" for thread pools $ts = \{o_1 \mapsto t_1, \ldots, o_n \mapsto t_n\}$:

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

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:

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

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

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

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

[5] `spec_public` can be defined in terms of class axioms, see [11].

6.2 Kripke Resource Semantics

We define a forcing relation of the form $\Gamma \vdash \mathscr{E}; \mathscr{R}; s \models F$, where Γ is a *type environment*, $\mathscr{E}$ is a *predicate environment*, $\mathscr{R}$ is a *resource*, and s is a *stack*. We assume that the stack s, the formula F, and the resource $\mathscr{R}$ are well-typed in Γ, i.e., the semantic relation is defined on well-typed tuples. The predicate environment $\mathscr{E}$ 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 $\mathscr{R}$ range over the set $\mathsf{Resource}$ with a binary relation $\# \subseteq \mathsf{Resource} \times \mathsf{Resource}$ (the *compatibility relation*) and a partial binary operator $* : \# \rightharpoonup \mathsf{Resource}$ (the *resource joining operator*) that is associative and commutative. Concretely, resources are 5-tuples $\mathscr{R} = (h, \mathscr{P}, \mathscr{L}, \mathscr{F}, \mathscr{I})$: a *heap* h, a *permission table* $\mathscr{P} \in \mathsf{ObjId} \times \mathsf{FieldId} \to [0,1]$, an *abstract lock table* $\mathscr{L} \in \mathsf{ObjId} \rightharpoonup \mathsf{Bag}(\mathsf{ObjId})$[6], a *fresh set* $\mathscr{F} \subseteq \mathsf{ObjId}$, and an *initialized set* $\mathscr{I} \subseteq \mathsf{ObjId}$. We require that resources satisfy the following axioms: *(1)* $\mathscr{P}(o,f) > 0$ iff $o \in \mathsf{dom}(h)$ and $f \in \mathsf{dom}(h(o)_2)$, *(2)* $\mathscr{F} \cap \mathscr{I} = \emptyset$, and *(3)* if $o \in \mathscr{L}(p)$ then $o \in \mathscr{I}$. Each of the five resource components carries itself a resource structure $(\#, *)$. Their structures are lifted to 5-tuples in a componentwise way. We now define $\#$ and $*$ for the five components.

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

$$h \# h' \text{ iff } \begin{cases} (\forall o \in \mathsf{dom}(h) \cap \mathsf{dom}(h'))(\\ h(o)_1 = h'(o)_1 \text{ and } (\forall f \in \mathsf{dom}(h(o)_2) \cap \mathsf{dom}(h'(o)_2))(\, h(o)_2(f) = h'(o)_2(f)\,) \;) \end{cases}$$

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

$$(h * h')(o)_1 \stackrel{\Delta}{=} h(o)_1 \vee h'(o)_1 \qquad (h * h')(o)_2 \stackrel{\Delta}{=} h(o)_2 \vee h'(o)_2$$

Joining *permission tables* is pointwise addition:

$$\mathscr{P} \# \mathscr{P}' \text{ iff } (\forall o)(\mathscr{P}(o) + \mathscr{P}'(o) \leq 1) \qquad (\mathscr{P} * \mathscr{P}')(o) \stackrel{\Delta}{=} \mathscr{P}(o) + \mathscr{P}'(o)$$

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

$$\mathscr{L} \# \mathscr{L}' \text{ iff } \begin{cases} \mathsf{dom}(\mathscr{L}) \cap \mathsf{dom}(\mathscr{L}') = \emptyset \\ (\forall o \in \mathsf{dom}(\mathscr{L}), p \in \mathsf{dom}(\mathscr{L}'))(\mathscr{L}(o) \sqcap \mathscr{L}'(p) = [\,]) \end{cases} \qquad \mathscr{L} * \mathscr{L}' \stackrel{\Delta}{=} \mathscr{L} \cup \mathscr{L}'$$

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

$$\begin{array}{lll} \mathscr{F} \# \mathscr{F}' & \text{iff} & \mathscr{F} \cap \mathscr{F}' = \emptyset \qquad \mathscr{F} * \mathscr{F}' \stackrel{\Delta}{=} \mathscr{F} \cup \mathscr{F}' \\ \mathscr{I} \# \mathscr{I}' & \text{iff} & \mathscr{I} = \mathscr{I}' \qquad \mathscr{I} * \mathscr{I}' \stackrel{\Delta}{=} \mathscr{I}\ (= \mathscr{I}') \end{array}$$

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: $[\![\texttt{nil}]\!]^h_s = [\,]$ and $[\![e \cdot e']\!]^h_s = [\![e]\!]^h_s \sqcup [\![e']\!]^h_s$. Here are the semantic clauses for our new formulas for reentrant locking:

[6] $\sqcap$ denotes bag intersection, $\sqcup$ bag union, and $[\,]$ the empty bag.

$$\begin{array}{lll}
\Gamma \vdash \mathscr{E}; (h, \mathscr{P}, \mathscr{L}, \mathscr{F}, \mathscr{I}); s \models \texttt{Lockset(}\pi\texttt{)} & \text{iff} & \mathscr{L}(o) = [\![\pi]\!] \text{ for some } o \\
\Gamma \vdash \mathscr{E}; (h, \mathscr{P}, \mathscr{L}, \mathscr{F}, \mathscr{I}); s \models \pi \texttt{ contains } e & \text{iff} & [\![e]\!]^h_s \in [\![\pi]\!] \\
\Gamma \vdash \mathscr{E}; (h, \mathscr{P}, \mathscr{L}, \mathscr{F}, \mathscr{I}); s \models e\texttt{.fresh} & \text{iff} & [\![e]\!]^h_s \in \mathscr{F} \\
\Gamma \vdash \mathscr{E}; (h, \mathscr{P}, \mathscr{L}, \mathscr{F}, \mathscr{I}); s \models e\texttt{.initialized} & \text{iff} & [\![e]\!]^h_s \in \mathscr{I}
\end{array}$$

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

6.3 Soundness

In this section, we extend our verification rules to runtime states. Of course, 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 $\mathscr{R} = (h, \mathscr{P}, \mathscr{L}, \mathscr{F}, \mathscr{I})$, let $\mathscr{R}_{\mathsf{hp}} = h$, $\mathscr{R}_{\mathsf{perm}} = \mathscr{P}$, $\mathscr{R}_{\mathsf{lock}} = \mathscr{L}$, $\mathscr{R}_{\mathsf{fresh}} = \mathscr{F}$ and $\mathscr{R}_{\mathsf{init}} = \mathscr{I}$. Our forcing relation $\models$ from the last section assumes formulas without logical variables: we deal with those by substitution, ranged over by $\sigma \in \mathsf{LogVar} \rightharpoonup \mathsf{SpecVal}$. We let $(\Gamma \vdash \sigma : \Gamma')$ whenever $\mathsf{dom}(\sigma) = \mathsf{dom}(\Gamma')$ and $(\Gamma[\sigma] \vdash \sigma(\alpha) : \Gamma'(\alpha)[\sigma])$ for all α in $\mathsf{dom}(\sigma)$. Furthermore, we let $\mathsf{cfv}(c) = \{x \in \mathsf{fv}(c) \mid x \text{ occurs in an object creation command } \ell\texttt{=new } C\texttt{<}\bar{\pi}\texttt{>}\}$.

Now, we extend the Hoare triple judgment to threads:

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

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:

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

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

$$\mathsf{ready}(\mathscr{R}) \stackrel{\Delta}{=} \mathscr{R}_{\mathsf{init}} \setminus \{o \mid (\exists p)(o \in \mathscr{L}(p))\}$$

$$\mathsf{conc}(\mathscr{L})(o) \stackrel{\Delta}{=} (p, \mathscr{L}(p)(o)), \text{ if } o \in \mathscr{L}(p) \qquad \mathsf{conc}(\mathscr{L})(o) \stackrel{\Delta}{=} \mathsf{free}, \text{ otherwise}$$

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

$$\frac{\begin{array}{c}h = (\mathscr{R} * \mathscr{R}')_{\mathsf{hp}} \qquad l = \mathsf{conc}(\mathscr{R}_{\mathsf{lock}}) \qquad \mathscr{R} \vdash ts : \diamond \\ \mathscr{R} \# \mathscr{R}' \qquad \mathscr{R}'_{\mathsf{lock}} = \emptyset \qquad \mathsf{fst} \circ \mathscr{R}'_{\mathsf{hp}} \subseteq \mathsf{fst} \circ h = \Gamma \qquad \Gamma \vdash \mathscr{E}; \mathscr{R}'; \emptyset \models \circledast_{o \in \mathsf{ready}(\mathscr{R})} o\texttt{.inv}\end{array}}{\langle h, l, ts \rangle : \diamond} \text{ (State)}$$

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. See Appendix Q for details.

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 [11] 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. [16] 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, Abraham, 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-parametrized 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 the overview section, but we have treated it in the appendix. 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 multi-threaded 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, 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.
12. C. Haack, C. Hurlin. Separation logic contracts for a Java-like language with fork/join. Technical Report 6430, INRIA, 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. A. Igarashi, B. Pierce, P. Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Trans. Program. Lang. Syst.*, 23(3), 2001.
15. S. Ishtiaq, P. W. O'Hearn. BI as an assertion language for mutable data structures. In *Principles of Programming Languages*, 2001.
16. 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.
17. P. Müller. *Modular Specification and Verification of Object-Oriented Programs*, vol. 2262 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.
18. M. Naftalin, P. Wadler. *Java Generics*. O'Reilly, 2006.

19. P. W. O'Hearn. Resources, concurrency and local reasoning. *Theoretical Computer Science*, 375(1–3), 2007.
20. P. W. O'Hearn, D. J. Pym. The logic of bunched implications. *Bulletin of Symbolic Logic*, 5(2), 1999.
21. M. Parkinson. *Local Reasoning for Java*. PhD thesis, University of Cambridge, 2005.
22. M. Parkinson, G. Bierman. Separation logic and abstraction. In *Principles of Programming Languages*, 2005.
23. J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Logic in Computer Science*, Copenhagen, Denmark, 2002. IEEE Press.
24. P. Wadler. A taste of linear logic. In *Mathematical Foundations of Computer Science*, 1993.

2

Technical Appendix

A Notational Conventions

We use the same syntax conventions as Featherweight Java (FJ) [14]. In particular, we indicate sequences of X's by an overbar: $\bar{X}$. We also use regular expression notation: X**?** for an optional X, X* for a possibly empty list of X's, $X \mid Y$ for an X or a Y, and $X\ Y$ for X followed by Y. For any syntactic object X, we let $\mathsf{fv}(X)$ be the set of free variables of X. We often write $x \notin X$ to abbreviate $x \notin \mathsf{fv}(X)$.

B Class Tables, Interfaces, Subtyping (Appendix for Section 2)

This appendix fills in the details that were omitted in Section 2.

The Thread class. We assume that class tables always contain the class `Thread`:

```
class Thread ext Object {
  pred preStart = true;
  final <lockset s> req preStart * Lockset(s); ens Lockset(s);
  void start();
  req preStart; ens true;
  void run() { null }
}
```

The `start`-method does not have a Java implementation, but is implemented natively. To model this, our operational semantics treats `start` in a special way. Intuitively, `start` behaves as follows:

- o`.start()` creates a new thread, whose thread identifier is o, and executes o`.run()` in this thread. The o`.start`-method should not be called more than once (on the same receiver o). If a thread p calls o`.start()` although o`.start()` has previously been called (by p or some other thread), then p blocks forever. In reality, a runtime exception is thrown in this case.

The `run`-method is meant to be overridden. The `preStart`-predicate is meant to be extended in subclasses. It is assumed as a precondition in the verification of `run`, and is required to be established by callers of `start`. In addition to `preStart`, the precondition of `start` requires `Lockset(s)`. This ensures that the precondition cannot be established if `preStart` depends on the `Lockset`-predicate. It is important to forbid such a dependency, because the `Lockset`-predicate implicitly depends on the current thread. Obviously, for the `start`-method the caller's current thread differs from the callee's current thread. Therefore a dependency of `preStart` on `Lockset` would not make sense.

The Object class. We assume that class tables always contain the class `Object`. This class is is the top element w.r.t. the subclass order $\preceq$, and thus has no `extends`-clause.

```
class Object {
  pred inv = true;
  final <lockset s>
  req Lockset(s) * s contains this * this.inv; ens Lockset(s) * this.inv;
  void wait();
  final <lockset s>
  req Lockset(s) * s contains this; ens Lockset(s);
  void notify();
}
```

The methods `wait` and `notify` do not have Java implementations, but are implemented natively. To model this, our operational semantics treats them in a special way. Intuitively, these methods behave as follows:

- If o.`wait()` is called when o is locked at reentrancy level n, then o's lock is released and the current thread p temporarily stops executing. When another thread calls o.`notify()`, thread p may be scheduled to resume execution and start competing for o's lock. When thread p reacquires o's lock, its reentrancy level is restored to n.

Note that the preconditions for `wait` and `notify` require that the receiver is locked. These requirements statically prevent `IllegalMonitorStateExceptions`, which are the runtime exceptions that Java throws when o.`wait()` or o.`notify()` are called without holding o's lock. The postcondition of o.`wait()` ensures o.`inv`, because o gets locked just before o.`wait()` terminates.

Interfaces are defined as follows:

pt ::= `pred` P`<`$\bar{T}\ \bar{\alpha}$`>;`	predicate types
mt ::= `<`$\bar{T}\ \bar{\alpha}$`>` $spec\ U\ m$`(`$\bar{V}\ \bar{\imath}$`);`	method types (scope of $\bar{\alpha}, \bar{\imath}$ is $\bar{T}, spec, U, \bar{V}$)
$int \in$ Interface ::= `interface` I`<`$\bar{T}\ \bar{\alpha}$`>ext` $\bar{U}$ `{`pt`*` mt`*}`	interfaces (scope of $\bar{\alpha}$ is $\bar{T}, \bar{U}, pt*, mt*$)

Subclassing. We use the symbol $\preceq_{ct}$ for the partial order on type identifiers induced by class table ct, usually leaving the subscript ct implicit. We impose the following sanity conditions on ct: *(1)* $\preceq_{ct}$ is antisymmetric, *(2)* if t occurs anywhere in ct then t is declared in ct, and *(3)* ct does not contain duplicate declarations.

Subtyping. The subtyping relation is mostly standard, apart from the following technicalities that are needed to deal with locksets.

- It is convenient to allow using objects as singleton locksets (rather than introducing explicit syntax for converting from objects to singleton locksets). Hence, we postulate `Object` $<:$ `lockset`.
- We allow arbitrary specification values (including locksets) as type parameters. While we are not sure if locksets as type parameters are useful in practice, they are naturally allowed by our system. In order for our type system to interact well with our logical axioms, we postulate that types with semantically equal type parameters are type-equivalent. Technically, we let $\simeq$ be the least equivalence relation on specification values that satisfies the standard multiset axioms:

$$\texttt{nil} \cdot \pi \simeq \pi \qquad \pi \cdot \pi' \simeq \pi' \cdot \pi \qquad (\pi \cdot \pi') \cdot \pi'' \simeq \pi \cdot (\pi' \cdot \pi'')$$

Then we postulate that $t\texttt{<}\bar{\pi}\texttt{>} <: t\texttt{<}\bar{\pi}'\texttt{>}$ when $\bar{\pi} \simeq \bar{\pi}'$.

In summary, subtyping $<:$ is inductively defined by the following rules:

$$T <: T \qquad T <: U, U <: V \;\Rightarrow\; T <: V \qquad s\texttt{<}\bar{T}\ \bar{\alpha}\texttt{>}\ \texttt{ext}\ t\texttt{<}\bar{\pi}'\texttt{>} \;\Rightarrow\; s\texttt{<}\bar{\pi}\texttt{>} <: t\texttt{<}\bar{\pi}'[\bar{\pi}/\bar{\alpha}]\texttt{>}$$
$$t\texttt{<}\bar{\pi}\texttt{>} <: \texttt{Object} \qquad t\texttt{<}\bar{T}\ \bar{\alpha}\texttt{>}\ \texttt{impl}\ I\texttt{<}\pi'\texttt{>} \;\Rightarrow\; t\texttt{<}\bar{\pi}\texttt{>} <: I\texttt{<}\bar{\pi}'[\bar{\pi}/\bar{\alpha}]\texttt{>}$$
$$\texttt{Object} <: \texttt{lockset} \qquad \bar{\pi} \simeq \bar{\pi}' \;\Rightarrow\; t\texttt{<}\bar{\pi}\texttt{>} <: t\texttt{<}\bar{\pi}'\texttt{>}$$

C Auxiliary Functions

Field Lookup, $\mathsf{fld}(C\texttt{<}\bar{\pi}\texttt{>}) = \bar{T}\ \bar{f}$**:**

(Fields Base)
$$\frac{}{\mathsf{fld}(\texttt{Object}) = \emptyset}$$

(Fields Ind)
$$\frac{\mathsf{fld}(D\texttt{<}\bar{\pi}'[\bar{\pi}/\bar{\alpha}]\texttt{>}) = \bar{T}'\ \bar{f}' \qquad \mathit{fin}\ \texttt{class}\ C\texttt{<}\bar{T}\ \bar{\alpha}\texttt{>}\ \texttt{ext}\ D\texttt{<}\bar{\pi}'\texttt{>}\ \texttt{impl}\ \bar{U}\ \texttt{\{}\bar{T}\ \bar{f}\ \mathit{pd}\texttt{*}\ \mathit{md}\texttt{*}\texttt{\}}}{\mathsf{fld}(C\texttt{<}\bar{\pi}\texttt{>}) = (\bar{T}\ \bar{f})[\bar{\pi}/\bar{\alpha}], \bar{T}'\ \bar{f}'}$$

Remarks on method lookup (defined below):

- In mbody and mtype, we replace the implicit self-parameter `this` by an explicit method parameter (separated from the other method parameters by a semicolon). This is technically convenient for the theory.
- In mtype, we replace the implicit result-parameter `result` by an explicit existential quantifier over the postcondition. This is technically convenient for the theory.

Method Lookup, $\mathsf{mtype}(m, t\texttt{<}\bar{\pi}\texttt{>}) = \mathit{fin}\ \mathit{mt}$ **and** $\mathsf{mbody}(m, C\texttt{<}\bar{\pi}\texttt{>}) = (\bar{\imath}).c$**:**

(Mlkup Object)
$$\frac{\texttt{class Object \{}\ldots\ \mathit{fin}\ \texttt{<}\bar{T}\ \bar{\alpha}\texttt{>}\ \mathit{spec}\ U\ m\texttt{(}\bar{V}\ \bar{\imath}\texttt{)\{}c\texttt{\}}\ \ldots\texttt{\}}}{\mathsf{mlkup}(m, \texttt{Object}) = \mathit{fin}\ \texttt{<}\bar{T}\ \bar{\alpha}\texttt{>}\ \mathit{spec}\ U\ m\texttt{(}\bar{V}\ \bar{\imath}\texttt{)\{}c\texttt{\}}}$$

(Mlkup Defn)
$$\frac{\mathit{fin}\ \texttt{class}\ C\texttt{<}\bar{T}'\ \bar{\alpha}'\texttt{>}\ \texttt{ext}\ U'\ \texttt{impl}\ \bar{V}'\ \texttt{\{}\ldots\ \mathit{fin}\ \texttt{<}\bar{T}\ \bar{\alpha}\texttt{>}\ \mathit{spec}\ U\ m\texttt{(}\bar{V}\ \bar{\imath}\texttt{)\{}c\texttt{\}}\ \ldots\texttt{\}}}{\mathsf{mlkup}(m, C\texttt{<}\bar{\pi}\texttt{>}) = (\mathit{fin}\ \texttt{<}\bar{T}\ \bar{\alpha}\texttt{>}\ \mathit{spec}\ U\ m\texttt{(}\bar{V}\ \bar{\imath}\texttt{)\{}c\texttt{\}})[\bar{\pi}/\bar{\alpha}']}$$

(Mlkup Inherit) $m \notin \mathsf{dom}(\mathit{md}\texttt{*})$
$$\frac{\mathit{fin}\ \texttt{class}\ C\texttt{<}\bar{T}\ \bar{\alpha}\texttt{>}\ \texttt{ext}\ D\texttt{<}\bar{\pi}'\texttt{>}\ \texttt{impl}\ \bar{U}\ \texttt{\{}\mathit{fd}\texttt{*}\ \mathit{pd}\texttt{*}\ \mathit{md}\texttt{*}\texttt{\}} \qquad \mathsf{mlkup}(m, D\texttt{<}\bar{\pi}'[\bar{\pi}/\bar{\alpha}]\texttt{>}) = \mathit{md}'}{\mathsf{mlkup}(m, C\texttt{<}\bar{\pi}\texttt{>}) = \mathit{md}'}$$

If $\mathsf{mlkup}(m, C\texttt{<}\bar{\pi}\texttt{>}) = \mathit{fin}\ \texttt{<}\bar{T}\ \bar{\alpha}\texttt{>}\ \texttt{req}\ F\texttt{;}\ \texttt{ens}\ G\texttt{;}\ U\ m\texttt{(}\bar{V}\ \bar{\imath}\texttt{)\{}c\texttt{\}}$, then:

$\mathsf{mbody}(m, C\texttt{<}\bar{\pi}\texttt{>}) \triangleq (\texttt{this;}\bar{\imath}).c$

$\mathsf{mtype}(m, C\texttt{<}\bar{\pi}\texttt{>}) \triangleq \mathit{fin}\ \texttt{<}\bar{T}\ \bar{\alpha}\texttt{>req}\ F\texttt{;ens (ex}\ U\ \texttt{result)(}G\texttt{);}\ U\ m\texttt{(}C\texttt{<}\bar{\pi}\texttt{>}\ \texttt{this;}\bar{V}\ \bar{\imath}\texttt{)}$

(Mtype Interface)
$$\frac{\texttt{interface}\ I\texttt{<}\bar{T}\ \bar{\alpha}\texttt{>ext}\ \bar{U}\ \texttt{\{}\ldots\ \texttt{<}\bar{T}'\ \bar{\alpha}'\texttt{>req}\ F\texttt{;ens}\ G\texttt{;}\ U'\ m\texttt{(}\bar{V}'\ \bar{\imath}\texttt{);}\ \ldots\texttt{\}}}{\mathsf{mtype}(m, I\texttt{<}\bar{\pi}\texttt{>}) = (\texttt{<}\bar{T}'\ \bar{\alpha}'\texttt{>req}\ F\texttt{;ens (ex}\ U'\ \texttt{result)(}G\texttt{);}U'\ m\texttt{(}I\texttt{<}\bar{\pi}\texttt{>}\ \texttt{this;}\ \bar{V}'\ \bar{\imath}\texttt{))}[\bar{\pi}/\bar{\alpha}]}$$

(Mtype Interface Inherit) $\texttt{interface}\ I\texttt{<}\bar{T}\ \bar{\alpha}\texttt{>ext}\ \bar{U}, V, \bar{U}'\ \texttt{\{}\mathit{pd}\texttt{*}\ \mathit{md}\texttt{*}\texttt{\}}$
$$\frac{m \notin \mathsf{dom}(\mathit{md}\texttt{*}) \quad (\forall U \in \bar{U}, \bar{U}')(\mathsf{mtype}(m, U[\bar{\pi}/\bar{\alpha}]) = \mathsf{undef}) \quad \mathsf{mtype}(m, V[\bar{\pi}/\bar{\alpha}]) = \mathit{fin}\ \mathit{mt}}{\mathsf{mtype}(m, I\texttt{<}\bar{\pi}\texttt{>}) = \mathit{fin}\ \mathit{mt}}$$

(Mtype Interface Inherit `Object`) $\quad$ `interface` $I\texttt{<}\bar{T}\ \bar{\alpha}\texttt{>}$ `ext` $\bar{U}$ `{`$pd\texttt{*}\ md\texttt{*}$`}`

$$\frac{m \notin \mathsf{dom}(md\texttt{*}) \quad (\forall U \in \bar{U})(\mathsf{mtype}(m, U[\bar{\pi}/\bar{\alpha}]) = \mathsf{undef}) \quad \mathsf{mtype}(m, \texttt{Object}) = \mathit{fin}\ mt}{\mathsf{mtype}(m, I\texttt{<}\bar{\pi}\texttt{>}) = \mathit{fin}\ mt}$$

Remarks on predicate lookup:

- The "ext `Object`" in the conclusion of (Plkup `Object`) and (Plkup `Object init`) is included to match the format of the relation. There is nothing more to this.
- Each class implicitly defines the `init`-predicate, which gives write permission to all fields of the class frame. In (Plkup `init`), $\mathsf{df}(T)$ is the default value of type T.

Predicate Lookup, $\mathsf{ptype}(P, t\texttt{<}\bar{\pi}\texttt{>}) = \mathit{fin}\ pt$ **and** $\mathsf{pbody}(\pi.P\texttt{<}\bar{\pi}'\texttt{>}, C\texttt{<}\bar{\pi}''\texttt{>}) = F$ `ext` T**:**

(Plkup `Object`)

$$\frac{\texttt{class Object \{}\ldots\ \mathit{fin}\ \texttt{pred}\ P\texttt{<}\bar{T}\ \bar{\alpha}\texttt{>=}F\texttt{;}\ \ldots\texttt{\}}}{\mathsf{plkup}(P, \texttt{Object}) = \mathit{fin}\ \texttt{pred}\ P\texttt{<}\bar{T}\ \bar{\alpha}\texttt{>=}F\ \texttt{ext Object}}$$

(Plkup `Object init`)

$$\frac{}{\mathsf{plkup}(\texttt{init}, \texttt{Object}) = \ \texttt{pred init=true ext Object}}$$

(Plkup Defn)

$$\frac{\mathit{fin}\ \texttt{class}\ C\texttt{<}\bar{T}'\ \bar{\alpha}'\texttt{>}\ \texttt{ext}\ U\ \texttt{impl}\ \bar{V}\ \texttt{\{}\ldots\ \mathit{fin}\ \texttt{pred}\ P\texttt{<}\bar{T}\ \bar{\alpha}\texttt{>=}F\texttt{;}\ \ldots\texttt{\}}}{\mathsf{plkup}(P, C\texttt{<}\bar{\pi}\texttt{>}) = (\mathit{fin}\ \texttt{pred}\ P\texttt{<}\bar{T}\ \bar{\alpha}\texttt{>=}F\ \texttt{ext Object})[\bar{\pi}/\bar{\alpha}']}$$

(Plkup `init`)

$$\frac{\mathit{fin}\ \texttt{class}\ C\texttt{<}\bar{T}'\ \bar{\alpha}'\texttt{>}\ \texttt{ext}\ U\ \texttt{impl}\ \bar{V}\ \texttt{\{}fd\texttt{*}\ pd\texttt{*}\ md\texttt{*}\texttt{\}} \quad F = \circledast_{T\ f \in fd\texttt{*}}\texttt{PointsTo(this.}f\texttt{,1,}\mathsf{df}(T)\texttt{)}}{\mathsf{plkup}(\texttt{init}, C\texttt{<}\bar{\pi}\texttt{>}) = (\ \texttt{pred init=}F\ \texttt{ext}\ U\ )[\bar{\pi}/\bar{\alpha}']}$$

(Plkup Inherit) $\quad P \notin \mathsf{dom}(pd\texttt{*})$

$$\frac{\mathit{fin}\ \texttt{class}\ C\texttt{<}\bar{T}'\ \bar{\alpha}'\texttt{>}\ \texttt{ext}\ U\ \texttt{impl}\ \bar{V}\ \texttt{\{}fd\texttt{*}\ pd\texttt{*}\ md\texttt{*}\texttt{\}} \quad \mathsf{plkup}(P, U) = \mathit{fin}\ \texttt{pred}\ P\texttt{<}\bar{T}\ \bar{\alpha}\texttt{>=}F\ \texttt{ext}\ U'}{\mathsf{plkup}(P, C\texttt{<}\bar{\pi}\texttt{>}) = (\mathit{fin}\ \texttt{pred}\ P\texttt{<}\bar{T}\ \bar{\alpha}\texttt{>=true ext}\ U)[\bar{\pi}/\bar{\alpha}']}$$

If $\mathsf{plkup}(P, C\texttt{<}\bar{\pi}\texttt{>}) = \mathit{fin}\ \texttt{pred}\ P\texttt{<}\bar{T}\ \bar{\alpha}\texttt{>=}F\ \texttt{ext}\ V$, then:

$$\mathsf{pbody}(\pi.P\texttt{<}\bar{\pi}'\texttt{>}, C\texttt{<}\bar{\pi}\texttt{>}) \triangleq (F\ \texttt{ext}\ V)[\pi/\texttt{this}, \bar{\pi}'/\bar{\alpha}]$$

$$\mathsf{ptype}(P, C\texttt{<}\bar{\pi}\texttt{>}) \triangleq \mathit{fin}\ \texttt{pred}\ P\texttt{<}\bar{T}\ \bar{\alpha}\texttt{>}$$

(Ptype Interface)

$$\frac{\texttt{interface}\ I\texttt{<}\bar{T}\ \bar{\alpha}\texttt{>ext}\ \bar{U}\ \texttt{\{}\ldots\ \texttt{pred}\ P\texttt{<}\bar{T}'\ \bar{\alpha}'\texttt{>;}\ \ldots\texttt{\}}}{\mathsf{ptype}(P, I\texttt{<}\bar{\pi}\texttt{>}) = (\texttt{pred}\ P\texttt{<}\bar{T}'\ \bar{\alpha}'\texttt{>})[\bar{\pi}/\bar{\alpha}]}$$

(Ptype Interface Inherit) $\quad$ `interface` $I\texttt{<}\bar{T}\ \bar{\alpha}\texttt{>ext}\ \bar{U}, V, \bar{U}'\ \texttt{\{}pd\texttt{*}\ md\texttt{*}\texttt{\}}$

$$\frac{P \notin \mathsf{dom}(pd\texttt{*}) \quad (\forall U \in \bar{U}, \bar{U}')(\mathsf{ptype}(P, U[\bar{\pi}/\bar{\alpha}]) = \mathsf{undef}) \quad \mathsf{ptype}(P, V[\bar{\pi}/\bar{\alpha}]) = \mathit{fin}\ pt}{\mathsf{ptype}(P, I\texttt{<}\bar{\pi}\texttt{>}) = \mathit{fin}\ pt}$$

(Ptype Interface Inherit `Object`) $\quad$ `interface` $I\texttt{<}\bar{T}\ \bar{\alpha}\texttt{>ext}\ \bar{U}\ \texttt{\{}pd\texttt{*}\ md\texttt{*}\texttt{\}}$

$$\frac{P \notin \mathsf{dom}(pd\texttt{*}) \quad (\forall U \in \bar{U})(\mathsf{ptype}(P, U[\bar{\pi}/\bar{\alpha}]) = \mathsf{undef}) \quad \mathsf{ptype}(P, \texttt{Object}) = \mathit{fin}\ pt}{\mathsf{ptype}(P, I\texttt{<}\bar{\pi}\texttt{>}) = \mathit{fin}\ pt}$$

D Type Environments and Types

A *type environment* is a partial function of type $\mathsf{ObjId} \cup \mathsf{Var} \rightharpoonup \mathsf{Type}$. We use the meta-variable Γ to range over type environments. Γ_{hp} denotes the *restriction of* Γ *to* ObjId:

$$\Gamma_{\mathsf{hp}} \triangleq \{ (o,T) \in \Gamma \mid o \in \mathsf{ObjId} \}$$

Good Environments, $\Gamma \vdash \diamond$:

(Env)
$$\frac{(\forall x \in \mathsf{dom}(\Gamma))(\Gamma \vdash \Gamma(x) : \diamond) \quad (\forall o \in \mathsf{dom}(\Gamma))(\Gamma(o) <: \texttt{Object} \text{ and } \Gamma_{\mathsf{hp}} \vdash \Gamma(o) : \diamond)}{\Gamma \vdash \diamond}$$

Good Types, $\Gamma \vdash T : \diamond$:

(Ty Primitive)
$$\frac{\Gamma \vdash \diamond \quad T \in \{\texttt{void}, \texttt{int}, \texttt{bool}, \texttt{perm}, \texttt{lockset}\}}{\Gamma \vdash T : \diamond}$$

(Ty Ref)
$$\frac{t\texttt{<}\bar{T}\ \bar{\alpha}\texttt{>} \in ct \quad \Gamma \vdash \diamond \quad \Gamma \vdash \bar{\pi} : \bar{T}[\bar{\pi}/\bar{\alpha}]}{\Gamma \vdash t\texttt{<}\bar{\pi}\texttt{>} : \diamond}$$

Note that types assigned to object ids cannot have free variables:

Lemma 1. *If* $(\Gamma \vdash \diamond)$*, then* $(\Gamma_{\mathsf{hp}} \vdash \diamond)$.

We define a *heap extension order* on well-formed type environments:

$$\Gamma' \supseteq_{\mathsf{hp}} \Gamma \quad \text{iff} \quad \Gamma' \vdash \diamond,\ \Gamma \vdash \diamond,\ \Gamma' \supseteq \Gamma \text{ and } \Gamma'_{|\mathsf{Var}} = \Gamma_{|\mathsf{Var}}$$

Semantics of types.

$$[\![\texttt{bool}]\!]^h \triangleq \{\texttt{true}, \texttt{false}\} \qquad [\![\texttt{int}]\!]^h = \mathsf{Int} \qquad [\![\texttt{Object}]\!]^h = \mathsf{dom}(h) \cup \{\texttt{null}\}$$
$$[\![\texttt{perm}]\!]^h = (0,1] \qquad [\![\texttt{lockset}]\!]^h = \mathsf{Bag}(\mathsf{ObjId}) \qquad [\![T_1, \ldots, T_n]\!]^h = [\![T_1]\!]^h \times \cdots \times [\![T_n]\!]^h$$

E Values and Specification Values

Values and specification values were defined in Section 2. Later in Section 4, we extended the syntax domain of specification values by bag operators. In summary, the syntax domains of values and specification values are defined like this:

$$n \in \mathsf{Int} \qquad u, v, w \in \mathsf{Val} ::= \texttt{null} \mid n \mid b \mid o \mid \iota$$
$$b \in \mathsf{Bool} = \{\texttt{true}, \texttt{false}\} \qquad \pi \in \mathsf{SpecVal} ::= \alpha \mid v \mid 1 \mid \texttt{split}(\pi) \mid \texttt{nil} \mid \pi \cdot \pi$$

Well-typed Values and Specification Values, $\Gamma \vdash v : T$ and $\Gamma \vdash \pi : T$:

(Val Var)
$$\frac{\Gamma \vdash \diamond \quad \Gamma(x) = T}{\Gamma \vdash x : T}$$

(Val Oid)
$$\frac{\Gamma \vdash \diamond \quad \Gamma(o) = T}{\Gamma \vdash o : T}$$

(Val Sub)
$$\frac{\Gamma \vdash \pi : T \quad T <: U}{\Gamma \vdash \pi : U}$$

(Val Null)
$$\frac{\Gamma \vdash t\texttt{<}\bar{\pi}\texttt{>} : \diamond}{\Gamma \vdash \texttt{null} : t\texttt{<}\bar{\pi}\texttt{>}}$$

(Val Int)
$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash n : \texttt{int}}$$

(Val Bool)
$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash b : \texttt{bool}}$$

(Val Full)
$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash 1 : \texttt{perm}}$$

(Val Split)
$$\frac{\Gamma \vdash \pi : \texttt{perm}}{\Gamma \vdash \texttt{split}(\pi) : \texttt{perm}}$$

(Val Nil)
$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash \texttt{nil} : \texttt{lockset}}$$

(Val Union)
$$\frac{\Gamma \vdash \pi : \texttt{lockset} \quad \Gamma \vdash \pi' : \texttt{lockset}}{\Gamma \vdash \pi \cdot \pi' : \texttt{lockset}}$$

Remarks on the value typing rules:

- Recall that `Object` $<:$ `lockset`. Thus, all object identifiers o have both type `Object` and type `lockset`. This reflects that we identify objects with singleton bags of objects.

Semantics of values. Recall that ClVal is the set of all closed values (excluding specification values). In other words, $\mathsf{ClVal} = \{\texttt{null}\} \cup \mathsf{ObjId} \cup \mathsf{Int} \cup \mathsf{Bool}$. We extend this set to include semantic domains for fractional permissions and locksets. The resulting set of *semantic values* is defined as follows:

$$\mu \in \mathsf{SemVal} \;\triangleq\; (\; \mathsf{ClVal} \cup (0,1] \cup \mathsf{Bag}(\mathsf{ObjId}) \;)/\equiv$$

where $\equiv$ is the least equivalence relation on $\mathsf{ClVal} \cup (0,1] \cup \mathsf{Bag}(\mathsf{ObjId})$ such that $o \equiv [o]$ for all object ids o. That is, $\equiv$ is the least equivalence relation that identifies object identifiers with singleton bags.

The following typing rules extend value typing to semantic values:

$$\frac{\mu \in (0,1]}{\Gamma \vdash \mu : \texttt{perm}} \qquad \frac{\mu \in \mathsf{Bag}(\mathsf{ObjId})}{\Gamma \vdash \mu : \texttt{lockset}}$$

Let $\mathsf{WtClSpecVal}$ be the the of well-typed, closed specification values:

$$\mathsf{WtClSpecVal} \triangleq \{\; \pi \mid (\exists \Gamma, T)(\mathsf{dom}(\Gamma) \subseteq \mathsf{ObjId} \text{ and } \Gamma \vdash \pi : T) \;\}$$

Now we define the semantics of specification values:

$$[\![\cdot]\!] : \mathsf{WtClSpecVal} \to \mathsf{SemVal}$$
$$[\![\texttt{null}]\!] \triangleq \texttt{null} \qquad [\![o]\!] \triangleq o \qquad [\![n]\!] \triangleq n \qquad [\![b]\!] \triangleq b$$
$$[\![1]\!] \triangleq 1 \qquad [\![\texttt{split}(\pi)]\!] \triangleq \tfrac{1}{2}[\![\pi]\!]$$
$$[\![\texttt{nil}]\!] \triangleq [\,] \qquad [\![\pi \cdot \pi']\!] \triangleq [\![\pi]\!] \sqcup [\![\pi']\!]$$

For our purposes, it is enough to define the semantics for closed values, because in our applications of the semantics we always first eliminate free read-only and logical variables by substitution.

Lemma 2 (Injectivity of Value Semantics). *If* $[\![\pi_1]\!] = [\![\pi_2]\!]$, *then* $\pi_1 \simeq \pi_2$.

F Expressions

In Section 3, we said that expressions e are built from values and variables using arithmetic and logical operators, as well as the operators e `instanceof` T and C `classof` e. More precisely, expressions are defined by the following grammar:

$$op \in \mathsf{Op} \;\supseteq\; \{\texttt{==}, \texttt{!}, \texttt{\&}, \texttt{|}\} \cup \{\, C \texttt{ classof} \mid C \in \mathsf{ClassId} \,\} \cup \{\, \texttt{instanceof } T \mid T \in \mathsf{Type} \,\}$$
$$e \in \mathsf{Exp} \;::=\; \pi \mid \ell \mid op(\vec{e})$$

We require that the operator set includes `==`, `!`, `&`, `|`, C `classof` , and `instanceof` T, because these operators are used in our proof rules and examples. Adding more operators to Op does not break our theory, as long as the operator semantics satisfies the axioms below.

Operator types. Let arity be a function that assigns to each operator its arity. We assume:

$$\mathsf{arity}(\texttt{==}) \triangleq \mathsf{arity}(\texttt{\&}) \triangleq \mathsf{arity}(\texttt{|}) \triangleq 2$$
$$\mathsf{arity}(\texttt{!}) \triangleq \mathsf{arity}(C\ \texttt{classof}) \triangleq \mathsf{arity}(\texttt{instanceof}\ T) \triangleq 1$$

Let type be a function that maps each operator op to a partial function $\mathsf{type}(op)$ of type $\{\texttt{int},\texttt{bool},\texttt{Object},\texttt{perm},\texttt{lockset}\}^{\mathsf{arity}(op)} \rightharpoonup \{\texttt{int},\texttt{bool},\texttt{perm},\texttt{lockset}\}$. We assume:

$$\mathsf{type}(\texttt{==}) \triangleq \{\ ((T,T),\texttt{bool}) \mid T \in \{\texttt{int},\texttt{bool},\texttt{Object},\texttt{perm},\texttt{lockset}\}\ \}$$
$$\mathsf{type}(\texttt{!}) \triangleq \{\ (\texttt{bool},\texttt{bool})\ \} \qquad \mathsf{type}(\texttt{\&}) \triangleq \mathsf{type}(\texttt{|}) \triangleq \{\ ((\texttt{bool},\texttt{bool}),\texttt{bool})\ \}$$
$$\mathsf{type}(C\ \texttt{classof}) \triangleq \{\ (\texttt{Object},\texttt{bool})\ \} \qquad \mathsf{type}(\texttt{instanceof}\ T) \triangleq \{\ (\texttt{Object},\texttt{bool})\ \}$$

Well-typed Expressions, $\Gamma \vdash e : T$:

(Exp Sub) $\dfrac{\Gamma \vdash e : T \quad T <: U}{\Gamma \vdash e : U}$ (Exp Val) $\dfrac{\Gamma \vdash \pi : T}{\Gamma \vdash \pi : T}$ (Exp Var) $\dfrac{\Gamma \vdash \diamond \quad \Gamma(\ell) = T}{\Gamma \vdash \ell : T}$ (Exp Op) $\dfrac{\Gamma \vdash \bar{e} : \bar{U} \quad \mathsf{type}(op)(\bar{U}) = T}{\Gamma \vdash op(\bar{e}) : T}$

Operator semantics. We assume that each operator op is interpreted by a function of the following type:

$$[\![op]\!] \ \in\ \mathsf{Heap} \to \bigcup_{(\bar{T},U)\in\mathsf{type}(op)} [\![\bar{T}]\!]^h \to [\![U]\!]^h$$

We require that the interpretation is closed under heap extensions, and that it does not depend on values in the heap:

(a) If $[\![op]\!]^h(\bar{v}) = w$ and $h \subseteq h'$, then $[\![op]\!]^{h'}(\bar{v}) = w$.
(b) If $h' = h[o.f \mapsto u]$, then $[\![op]\!]^h = [\![op]\!]^{h'}$.

For the logical operators `!`, `|` and `&`, we assume the usual interpretations. `==` is interpreted as the identity relation. $[\![C\ \texttt{classof}]\!]^h(o) = \texttt{true}$ when $h(o)_1 = C\texttt{<}\bar{\pi}\texttt{>}$ for some $\bar{\pi}$, $[\![C\ \texttt{classof}]\!]^h(o) = \texttt{false}$ when $h(o)_1 = D\texttt{<}\bar{\pi}\texttt{>}$ and $D \neq C$, $[\![C\ \texttt{classof}]\!]^h(o)$ undefined when $o \notin \mathsf{dom}(h)$, $[\![C\ \texttt{classof}]\!]^h(\texttt{null}) = \texttt{false}$. $[\![T\ \texttt{instanceof}\ ]\!]^h(o) = \texttt{true}$ when $h(o)_1 <: T$, $[\![T\ \texttt{instanceof}\ ]\!]^h(o) = \texttt{false}$ when $o \in \mathsf{dom}(h)$ and $h(o)_1 \not<: T$, $[\![T\ \texttt{instanceof}\ ]\!]^h(o)$ undefined when $o \notin \mathsf{dom}(h)$, $[\![T\ \texttt{instanceof}\ ]\!]^h(\texttt{null}) = \texttt{false}$.

Note that the interpretations of "$\texttt{instanceof}\ T$" and "$C\ \texttt{classof}$" depend on the heap, but still satisfy requirement (b), as they only depend on the type components of objects.

Semantics of Expressions, $[\![e]\!] : \mathsf{Heap} \to \mathsf{Stack} \rightharpoonup \mathsf{SemVal}$:

(Sem Val) $\dfrac{[\![\pi]\!] = \mu}{[\![\pi]\!]^h_s = \mu}$ (Sem Var) $\dfrac{s(\ell) = v}{[\![\ell]\!]^h_s = v}$ (Sem Op) $\dfrac{[\![e_1]\!]^h_s = \mu_1 \quad \cdots \quad [\![e_n]\!]^h_s = \mu_n \quad [\![op]\!]^h(\mu_1,\ldots,\mu_n) = \mu}{[\![op(e_1,\ldots,e_n)]\!]^h_s = \mu}$

G Formulas

Formulas were defined in Section 3 and extended in Section 4. In summary, the syntax domain of formulas is defined like this:

$$lop \in \{\texttt{*}, \texttt{-*}, \texttt{\&}, \texttt{|}\} \qquad qt \in \{\texttt{ex}, \texttt{fa}\} \qquad \kappa \in \mathsf{Pred} ::= P \mid P\texttt{@}C$$
$$F \in \mathsf{Formula} ::= e \mid \texttt{PointsTo(}e.f, \pi, e\texttt{)} \mid \pi.\kappa\texttt{<}\bar{\pi}\texttt{>} \mid F\ lop\ F \mid \texttt{(}qt\ T\ \alpha\texttt{)(}F\texttt{)} \mid \texttt{Lockset(}\pi\texttt{)} \mid \pi\ \texttt{contains}\ e \mid e\texttt{.fresh} \mid e\texttt{.initialized}$$

Well-typed formulas. As an auxiliary, we extend the partial function $\mathsf{ptype}(P, t\texttt{<}\bar{\pi}\texttt{>})$ to predicate selectors $P\texttt{@}C$:

$$\mathsf{ptype}(P\texttt{@}C, t\texttt{<}\bar{\pi}\texttt{>}) \triangleq \begin{cases} \mathsf{ptype}(P, t\texttt{<}\bar{\pi}\texttt{>}) & \text{if } t = C \\ \mathsf{undef} & \text{otherwise} \end{cases}$$

Well-typed Formulas, $\Gamma \vdash F : \diamond$:

(Form Bool)
$$\frac{\Gamma \vdash e : \texttt{bool}}{\Gamma \vdash e : \diamond}$$

(Form Points To)
$$\frac{\Gamma \vdash e : U \quad \Gamma \vdash \pi : \texttt{perm} \quad T\ f \in \mathsf{fld}(U) \quad \Gamma \vdash e' : T}{\Gamma \vdash \texttt{PointsTo(}e.f, \pi, e'\texttt{)} : \diamond}$$

(Form Log Op)
$$\frac{\Gamma \vdash F, F' : \diamond}{\Gamma \vdash F\ lop\ F' : \diamond}$$

(Form Pred)
$$\frac{\Gamma \vdash \pi : U \quad \mathsf{ptype}(\kappa, U) = \mathit{fin}\ \texttt{pred}\ P\texttt{<}\bar{T}\ \bar{\alpha}\texttt{>} \quad \Gamma \vdash \bar{\pi}' : \bar{T}}{\Gamma \vdash \pi.\kappa\texttt{<}\bar{\pi}'\texttt{>} : \diamond}$$

(Form Quant)
$$\frac{\Gamma \vdash T : \diamond \quad \Gamma, \alpha : T \vdash F : \diamond}{\Gamma \vdash \texttt{(}qt\ T\ \alpha\texttt{)(}F\texttt{)} : \diamond}$$

(Form Lockset)
$$\frac{\Gamma \vdash \pi : \texttt{lockset}}{\Gamma \vdash \texttt{Lockset(}\pi\texttt{)} : \diamond}$$

(Form Contains)
$$\frac{\Gamma \vdash \pi : \texttt{lockset,Object}}{\Gamma \vdash \pi\ \texttt{contains}\ e : \diamond}$$

(Form Fresh)
$$\frac{\Gamma \vdash e : \texttt{Object}}{\Gamma \vdash e\texttt{.fresh} : \diamond}$$

(Form Initialized)
$$\frac{\Gamma \vdash e : \texttt{Object}}{\Gamma \vdash e\texttt{.initialized} : \diamond}$$

Semantics of formulas. The semantics of formulas will be presented in Appendix N.

H Objects, Heaps, Stacks

Well-typed Objects, $\Gamma \vdash obj : \diamond$:

(Obj)
$$\frac{\mathsf{dom}(os) \subseteq \mathsf{dom}(\mathsf{fld}(C\texttt{<}\bar{\pi}\texttt{>})) \quad \Gamma \vdash C\texttt{<}\bar{\pi}\texttt{>} : \diamond \quad (\forall f \in \mathsf{dom}(os))(T\ f \in \mathsf{fld}(C\texttt{<}\bar{\pi}\texttt{>}) \Rightarrow \Gamma \vdash os(f) : T)}{\Gamma \vdash (C\texttt{<}\bar{\pi}\texttt{>}, os) : \diamond}$$

Note that we require $\mathsf{dom}(os) \subseteq \mathsf{dom}(\mathsf{fld}(C\texttt{<}\bar{\pi}\texttt{>}))$, not $\mathsf{dom}(os) = \mathsf{dom}(\mathsf{fld}(C\texttt{<}\bar{\pi}\texttt{>}))$. Thus, we allow partial objects. This is needed, because $*$ joins heaps on a per-field basis.

Well-typed Heaps and Stacks, $\Gamma \vdash h : \diamond$ and $\Gamma \vdash s : \diamond$:

(Heap)
$$\frac{\Gamma \vdash \diamond \quad \Gamma \subseteq \mathsf{fst} \circ h \quad (\forall o \in \mathsf{dom}(h))(\Gamma \vdash h(o) : \diamond)}{\Gamma \vdash h : \diamond}$$

(Stack)
$$\frac{\Gamma \vdash \diamond \quad (\forall x \in \mathsf{dom}(s))(\Gamma \vdash s(x) : \Gamma(x))}{\Gamma \vdash s : \diamond}$$

I Proof Theory

As usual, Hoare triples will be based on a logical consequence judgment. We define logical consequence proof-theoretically. The proof theory has two judgments:

$\Gamma; v; \bar{F} \vdash G$	G is a logical consequence of the $*$-conjunction of $\bar{F}$
$\Gamma; v \vdash F$	F is an axiom

In the former judgment, $\bar{F}$ is a *multiset* of formulas. The parameter v represents the *current receiver*. The receiver parameter is needed to determine the scope of predicate definitions: a receiver v knows the definitions of predicates of the form $v.P$, but not the definitions of other predicates. In source code verification, the receiver parameter is always `this` and can thus be omitted. We explicitly include the receiver parameter in the general judgment, because we want the proof theory to be closed under value substitutions.

Semantic validity of boolean expressions. The proof theory depends on the relation $\Gamma \models e$ ("e is valid in all well-typed heaps"), which we do not axiomatize. To define this relation, let σ range over *closing substitutions*, i.e, elements of $\mathsf{Var} \rightharpoonup \mathsf{ClSpecValSet}$.

$$\frac{\mathsf{dom}(\sigma) = \mathsf{dom}(\Gamma) \cap \mathsf{Var} \quad (\forall x \in \mathsf{dom}(\sigma))(\Gamma_{\mathsf{hp}} \vdash \sigma(x) : \Gamma(x)[\sigma])}{\Gamma \vdash \sigma : \diamond}$$

$$\mathsf{ClosingSubst}(\Gamma) \stackrel{\Delta}{=} \{\ \sigma \mid \Gamma \vdash \sigma : \diamond\ \}$$

We say that a heap h is *total* iff for all o in $\mathsf{dom}(h)$ and all $f \in \mathsf{dom}(\mathsf{fld}(h(o)_1))$ it is the case that $f \in \mathsf{dom}(h(o)_2)$.

$$\mathsf{Heap}(\Gamma) \stackrel{\Delta}{=} \{\ h \mid \Gamma_{\mathsf{hp}} \vdash h : \diamond \text{ and } h \text{ is total}\ \}$$

Now, we define $\Gamma \models e$ as follows:

$$\Gamma \models e \quad \text{iff} \quad \begin{cases} \Gamma \vdash e : \texttt{bool} \text{ and} \\ (\ \forall \Gamma' \supseteq_{\mathsf{hp}} \Gamma, h \in \mathsf{Heap}(\Gamma'), \sigma \in \mathsf{ClosingSubst}(\Gamma')\ )(\ [\![e[\sigma]]\!]^h_\emptyset = \texttt{true}\ ) \end{cases}$$

I.1 Natural Deduction Rules

The proof theory is driven by the natural deduction rules for (affine) linear logic:

Logical Consequence, $\Gamma; v; \bar{F} \vdash G$:

(Id)
$$\frac{\Gamma \vdash v, \bar{F}, G : \texttt{Object}, \diamond}{\Gamma; v; \bar{F}, G \vdash G}$$

(Ax)
$$\frac{\Gamma; v \vdash G \quad \Gamma \vdash v, \bar{F}, G : \texttt{Object}, \diamond}{\Gamma; v; \bar{F} \vdash G}$$

($*$ Intro)
$$\frac{\Gamma; v; \bar{F} \vdash H_1 \quad \Gamma; v; \bar{G} \vdash H_2}{\Gamma; v; \bar{F}, \bar{G} \vdash H_1 * H_2}$$

($*$ Elim)
$$\frac{\Gamma; v; \bar{F} \vdash G_1 * G_2 \quad \Gamma; v; \bar{E}, G_1, G_2 \vdash H}{\Gamma; v; \bar{F}, \bar{E} \vdash H}$$

(-$*$ Intro)
$$\frac{\Gamma; v; \bar{F}, G_1 \vdash G_2}{\Gamma; v; \bar{F} \vdash G_1 \mathrel{-\!\!*} G_2}$$

(-$*$ Elim)
$$\frac{\Gamma; v; \bar{F} \vdash H_1 \mathrel{-\!\!*} H_2 \quad \Gamma; v; \bar{G} \vdash H_1}{\Gamma; v; \bar{F}, \bar{G} \vdash H_2}$$

(& Intro)
$$\frac{\Gamma; v; \bar{F} \vdash G_1 \quad \Gamma; v; \bar{F} \vdash G_2}{\Gamma; v; \bar{F} \vdash G_1 \,\&\, G_2}$$

(& Elim 1)
$$\frac{\Gamma; v; \bar{F} \vdash G_1 \,\&\, G_2}{\Gamma; v; \bar{F} \vdash G_1}$$

(& Elim 2)
$$\frac{\Gamma; v; \bar{F} \vdash G_1 \,\&\, G_2}{\Gamma; v; \bar{F} \vdash G_2}$$

(| Intro 1)
$$\frac{\Gamma; v; \bar{F} \vdash G_1}{\Gamma; v; \bar{F} \vdash G_1 \mid G_2}$$

(| Intro 2)
$$\frac{\Gamma; v; \bar{F} \vdash G_2}{\Gamma; v; \bar{F} \vdash G_1 \mid G_2}$$

(| Elim) $\Gamma; v; \bar{F} \vdash G_1 \mid G_2$
$$\frac{\Gamma; v; \bar{E}, G_1 \vdash H \quad \Gamma; v; \bar{E}, G_2 \vdash H}{\Gamma; v; \bar{F}, \bar{E} \vdash H}$$

$$\text{(Ex Intro)}\quad \frac{\Gamma, \alpha : T \vdash G : \diamond \qquad \Gamma \vdash \pi : T \quad \Gamma; v; \bar{F} \vdash G[\pi/\alpha]}{\Gamma; v; \bar{F} \vdash \texttt{(ex}\ T\ \alpha\texttt{)(}G\texttt{)}} \qquad \text{(Ex Elim)}\quad \frac{\alpha \notin \bar{F}, H \qquad \Gamma; v; \bar{E} \vdash \texttt{(ex}\ T\ \alpha\texttt{)(}G\texttt{)} \quad \Gamma, \alpha : T; v; \bar{F}, G \vdash H}{\Gamma; v; \bar{E}, \bar{F} \vdash H}$$

$$\text{(Fa Intro)}\quad \frac{\alpha \notin \bar{F} \quad \Gamma, \alpha : T; v; \bar{F} \vdash G}{\Gamma; v; \bar{F} \vdash \texttt{(fa}\ T\ \alpha\texttt{)(}G\texttt{)}} \qquad \text{(Fa Elim)}\quad \frac{\Gamma; v; \bar{F} \vdash \texttt{(fa}\ T\ \alpha\texttt{)(}G\texttt{)} \quad \Gamma \vdash \pi : T}{\Gamma; v; \bar{F} \vdash G[\pi/\alpha]}$$

I.2 Axioms

First, we repeat the four axioms that we presented in Section 3 in their general form with explicit receiver parameter:

$$\Gamma; v \vdash \texttt{PointsTo(}e.f, \pi, e'\texttt{)} \mathrel{\texttt{*-*}} \texttt{(PointsTo(}e.f, \tfrac{\pi}{2}, e'\texttt{)} \mathbin{\texttt{*}} \texttt{PointsTo(}e.f, \tfrac{\pi}{2}, e'\texttt{))}$$

$$\begin{array}{c}(\Gamma \vdash v : C\texttt{<}\bar{\pi}''\texttt{>} \wedge \ \texttt{pbody}(v.P\texttt{<}\bar{\pi}, \bar{\pi}'\texttt{>}, C\texttt{<}\bar{\pi}''\texttt{>}) = F \ \texttt{ext}\ D\texttt{<}\bar{\pi}'''\texttt{>}) \\ \Rightarrow\ \Gamma; v \vdash v.P\texttt{@}C\texttt{<}\bar{\pi}, \bar{\pi}'\texttt{>} \mathrel{\texttt{*-*}} \texttt{(}F \mathbin{\texttt{*}} v.P\texttt{@}D\texttt{<}\bar{\pi}\texttt{>)}\end{array}$$

$$\Gamma; v \vdash \pi.P\texttt{<}\bar{\pi}\texttt{>} \mathrel{\texttt{*-*}} \texttt{(ex}\ \bar{T}\ \bar{\alpha}\texttt{)(}\pi.P\texttt{<}\bar{\pi}, \bar{\alpha}\texttt{>)}$$

$$\Gamma; v \vdash \pi.P\texttt{@}C\texttt{<}\bar{\pi}\texttt{>}\ \texttt{ispartof}\ \pi.P\texttt{<}\bar{\pi}\texttt{>}$$

There is another similar axiom:

$$C \preceq D \ \Rightarrow\ \Gamma; v \vdash \pi.P\texttt{@}D\texttt{<}\bar{\pi}\texttt{>}\ \texttt{ispartof}\ \pi.P\texttt{@}C\texttt{<}\bar{\pi}, \bar{\pi}'\texttt{>}$$

Our semantics of predicates defines predicates with `null`-receiver to hold:

$$\Gamma; v \vdash \texttt{null.}\kappa\texttt{<}\bar{\pi}\texttt{>}$$

The following axioms allow to drop the class modifier C from $\pi.P\texttt{@}C$ if we know that C is π's dynamic class, or if P is `final` in C:

$$\Gamma; v \vdash \texttt{(}\ \pi.P\texttt{@}C\texttt{<}\bar{\pi}\texttt{>} \mathbin{\texttt{*}} C\ \texttt{classof}\ \pi\ \texttt{)} \mathrel{\texttt{-*}} \pi.P\texttt{<}\bar{\pi}\texttt{>}$$

$$(C \text{ is } \texttt{final} \text{ or } P \text{ is } \texttt{final} \text{ in } C) \ \Rightarrow\ \Gamma; v \vdash \pi.P\texttt{@}C\texttt{<}\bar{\pi}\texttt{>} \mathrel{\texttt{-*}} \pi.P\texttt{<}\bar{\pi}\texttt{>}$$

Axiomatizing equality of specification values is standard:

$$\Gamma; v \vdash \pi \mathbin{\texttt{==}} \pi \qquad \Gamma; v \vdash \pi \mathbin{\texttt{==}} \pi' \ \Rightarrow\ \Gamma; v \vdash \pi' \mathbin{\texttt{==}} \pi$$

$$\Gamma; v \vdash \pi \mathbin{\texttt{==}} \pi',\ \Gamma; v \vdash \pi' \mathbin{\texttt{==}} \pi'' \ \Rightarrow\ \Gamma; v \vdash \pi \mathbin{\texttt{==}} \pi''$$

Recall that $\simeq$ is bag equality of specification values (see Appendix B for the definition). We lift bag equality to our proof theory, and axiomatize bag membership:

$$\pi \simeq \pi' \ \Rightarrow\ \Gamma; v \vdash \pi \mathbin{\texttt{==}} \pi'$$

$$\Gamma; v \vdash \texttt{!(nil contains}\ e\texttt{)} \qquad \Gamma; v \vdash \texttt{(}\pi \cdot \pi'\texttt{)}\ \texttt{contains}\ e \mathrel{\texttt{*-*}} \texttt{(}e \mathbin{\texttt{==}} \pi \ \texttt{|}\ \pi'\ \texttt{contains}\ e\texttt{)}$$

We axiomatize `true` and `false`:

$$\Gamma; v \vdash \texttt{true} \qquad \Gamma; v \vdash \texttt{false} \mathrel{\texttt{-*}} F$$

The substitutivity axiom allows to replace expressions by equal expressions:

$$(\Gamma \vdash e, e' : T \ \wedge\ \Gamma, x : T \vdash F : \diamond) \ \Rightarrow\ \Gamma; v \vdash \texttt{(}F[e/x] \mathbin{\texttt{*}} e \mathbin{\texttt{==}} e'\texttt{)} \mathrel{\texttt{-*}} F[e'/x]$$

An axiom lifts semantic validity of boolean expressions to the proof theory:

$$(\Gamma \models \texttt{!}e_1 \ \texttt{|}\ \texttt{!}e_2 \ \texttt{|}\ e') \ \Rightarrow\ \Gamma; v \vdash \texttt{(}e_1 \mathbin{\texttt{*}} e_2\texttt{)} \mathrel{\texttt{-*}} e'$$

The following axiom captures that fields point to a unique value. We write "F `assures` G" to abbreviate "F `-*` `(`$F * G$`)`".

$$\Gamma; v \vdash \texttt{(PointsTo(}e.f, \pi, e'\texttt{) \& PointsTo(}e.f, \pi', e''\texttt{)) assures } e' \texttt{ == } e''$$

There is an axiom that captures that all well-typed closed expressions represent a value (because built-in operations are total):

$$(\Gamma \vdash e : T) \;\Rightarrow\; \Gamma; v \vdash \texttt{(ex } T\ \alpha\texttt{)(}e \texttt{ == } \alpha\texttt{)}$$

Finally, there is a copyability axiom:

$$G \in \{e,\ \pi \texttt{ contains } e,\ e\texttt{.initialized}\} \;\Rightarrow\; \Gamma; v \vdash \texttt{(}F \texttt{ \& } G\texttt{) -* (}F * G\texttt{)}$$

J Hoare Triples

Our judgment for *commands* combines typing and Hoare triples:

$$\Gamma; v \vdash \{F\} c : T \{G\}$$

T is a type of the return value (possibly a supertype of the return value's dynamic type). G is the postcondition and is always of the form $G = \texttt{(ex } U\ \alpha\texttt{)(}G'\texttt{)}$ with $U <: T$. The existentially quantified α represents the return value.

Hoare triples for *head commands* have the following form:

$$\Gamma; v \vdash \{F\} hc \{G\}$$

Recall that $\mathsf{fv}(F)$ is the set of free variables of F, and that we write $x \notin F$ to abbreviate $x \notin \mathsf{fv}(F)$. Furthermore, let $\mathsf{writes}(hc)$ be the set of read-write variables ℓ that occur freely on the left-hand-side of an assignment in hc.

Hoare Triples for Commands, $\Gamma; v \vdash \{F\} c : T \{G\}$:

(Val)
$$\frac{\Gamma; v; F \vdash G[w/\alpha] \quad \Gamma \vdash w : U <: T \quad \Gamma, \alpha : U \vdash G : \diamond}{\Gamma; v \vdash \{F\} w : T \{\texttt{(ex } U\ \alpha\texttt{)(}G\texttt{)}\}}$$

(Dcl) $\ell \notin F, G$
$$\frac{\Gamma, \ell : T; v \vdash \{F * \ell \texttt{ == } \mathsf{df}(T)\} c : U \{G\}}{\Gamma; v \vdash \{F\} T\ \ell; c : U \{G\}}$$

(Fin Dcl) $\iota \notin F, G, v$
$$\frac{\Gamma \vdash \ell : T \quad \Gamma, \iota : T; v \vdash \{F * \iota \texttt{ == } \ell\} c : U \{G\}}{\Gamma; v \vdash \{F\} \texttt{final } T\ \iota \texttt{=} \ell; c : U \{G\}}$$

(Seq)
$$\frac{\Gamma; v \vdash \{F\} hc \{F'\} \quad \Gamma; v \vdash \{F'\} c : T \{G\}}{\Gamma; v \vdash \{F\} hc; c : T \{G\}}$$

Hoare Triples for Head Commands, $\Gamma; v \vdash \{F\} hc \{G\}$:

(Frame)
$$\frac{\Gamma; v \vdash \{F\} hc \{G\} \quad \Gamma \vdash H : \diamond \quad \mathsf{fv}(H) \cap \mathsf{writes}(hc) = \emptyset}{\Gamma; v \vdash \{F * H\} hc \{G * H\}}$$

(Con)
$$\frac{\Gamma; v; F \vdash F' \quad \Gamma; v \vdash \{F'\} hc \{G'\} \quad \Gamma; v; G' \vdash G}{\Gamma; v \vdash \{F\} hc \{G\}}$$

(Exists)
$$\frac{\Gamma, \alpha : T; v \vdash \{F\} hc \{G\}}{\Gamma; v \vdash \{\texttt{(ex } T\ \alpha\texttt{)(}F\texttt{)}\} hc \{\texttt{(ex } T\ \alpha\texttt{)(}G\texttt{)}\}}$$

(Disj)
$$\frac{\Gamma; v \vdash \{F\} hc \{G\} \quad \Gamma; v \vdash \{F'\} hc \{G'\}}{\Gamma; v \vdash \{F \mid F'\} hc \{G \mid G'\}}$$

(Var Set)
$$\frac{\Gamma \vdash v, w : \texttt{Object}, \Gamma(\ell)}{\Gamma; v \vdash \{\texttt{true}\} \ell \texttt{=} w \{\ell \texttt{ == } w\}}$$

(Op)
$$\frac{\Gamma \vdash v, op(\bar{w}) : \texttt{Object}, \Gamma(\ell)}{\Gamma; v \vdash \{\texttt{true}\} \ell \texttt{=} op(\bar{w}) \{\ell \texttt{ == } op(\bar{w})\}}$$

(Get)

$$\frac{\Gamma \vdash v,u,\pi,w : \texttt{Object}, U, \texttt{perm}, W \quad W\, f \in \mathsf{fld}(U) \quad W <: \Gamma(\ell)}{\Gamma; v \vdash \{\texttt{PointsTo}(u.f,\pi,w)\}\ell\texttt{=}u.f\{\texttt{PointsTo}(u.f,\pi,w) * \ell \texttt{ == } w\}}$$

(Fld Set)

$$\frac{\Gamma \vdash u,w : U, W \quad W\, f \in \mathsf{fld}(U)}{\Gamma; v \vdash \{\texttt{PointsTo}(u.f,1,W)\}u.f\texttt{=}w\{\texttt{PointsTo}(u.f,1,w)\}}$$

(New)

$$\frac{\Gamma \vdash v : \texttt{Object} \quad C\texttt{<}\bar{T}\ \bar{\alpha}\texttt{>} \in ct \quad \Gamma \vdash \bar{\pi} : \bar{T}[\bar{\pi}/\alpha] \quad C\texttt{<}\bar{\pi}\texttt{>} <: \Gamma(\ell)}{\Gamma; v \vdash \{\texttt{true}\}\ell\texttt{=new } C\texttt{<}\bar{\pi}\texttt{>}\{\ell.\texttt{init} * C\ \texttt{classof}\ \ell * \circledast_{\Gamma(u)<:\texttt{Object}}\ \ell\ \texttt{!=}\ u * \ell.\texttt{fresh}\}}$$

(Call)

$$\frac{\begin{array}{c}\mathsf{mtype}(m,t\texttt{<}\bar{\pi}\texttt{>}) = \mathit{fin}\ \texttt{<}\bar{T}\ \bar{\alpha}\texttt{>req}\, G; \texttt{ens (ex}\ U\ \alpha')\texttt{(}G'\texttt{)};\ U\, m(t\texttt{<}\bar{\pi}\texttt{>}\iota_0; \bar{W}\,\bar{\iota}) \\ \sigma = (u/\iota_0, \bar{\pi}'/\bar{\alpha}, \bar{w}/\bar{\iota}) \quad \Gamma \vdash u, \bar{\pi}', \bar{w} : t\texttt{<}\bar{\pi}\texttt{>}, \bar{T}[\sigma], \bar{W}[\sigma] \quad U[\sigma] <: \Gamma(\ell)\end{array}}{\Gamma; v \vdash \{u\ \texttt{!=null} * G[\sigma]\}\ell\texttt{=}u.m(\bar{w})\{\texttt{(ex}\ U[\sigma]\ \alpha')\texttt{(}\alpha' \texttt{ == } \ell * G'[\sigma]\texttt{)}\}}$$

(If)

$$\frac{\Gamma \vdash w : \texttt{bool} \quad \Gamma; v \vdash \{F * w\}c : \texttt{void}\{G\} \quad \Gamma; v \vdash \{F * \texttt{!}w\}c' : \texttt{void}\{G\}}{\Gamma; v \vdash \{F\}\texttt{if (}w\texttt{)\{}c\texttt{\}else\{}c'\texttt{\}}\{G\}}$$

(Assert)

$$\frac{\Gamma; v; F \vdash G}{\Gamma; v \vdash \{F\}\texttt{assert(}G\texttt{)}\{F\}}$$

(Commit)

$$\frac{F = \texttt{Lockset(}\pi\texttt{)} \quad \Gamma \vdash v,\pi,\pi' : \texttt{Object}, \texttt{lockset}, \texttt{Object}}{\Gamma; v \vdash \{F * \pi'.\texttt{inv} * \pi'.\texttt{fresh}\}\pi'.\texttt{commit}\{F * \texttt{!(}\pi\ \texttt{contains}\ \pi'\texttt{)} * \pi'.\texttt{initialized}\}}$$

(Lock)

$$\frac{F = \texttt{Lockset(}\pi\texttt{)} \quad \Gamma \vdash v,\pi,w : \texttt{Object}, \texttt{lockset}, \texttt{Object}}{\Gamma; v \vdash \{F * \texttt{!(}\pi\ \texttt{contains}\ w\texttt{)} * w.\texttt{initialized}\}w.\texttt{lock()}\{\texttt{Lockset(}w \cdot \pi\texttt{)} * w.\texttt{inv}\}}$$

(Re-Lock)

$$\frac{\Gamma \vdash v,\pi,w : \texttt{Object}, \texttt{lockset}, \texttt{Object}}{\Gamma; v \vdash \{\texttt{Lockset(}w \cdot \pi\texttt{)}\}w.\texttt{lock()}\{\texttt{Lockset(}w \cdot w \cdot \pi\texttt{)}\}}$$

(Re-Unlock)

$$\frac{\Gamma \vdash v,\pi,w : \texttt{Object}, \texttt{lockset}, \texttt{Object}}{\Gamma; v \vdash \{\texttt{Lockset(}w \cdot w \cdot \pi\texttt{)}\}w.\texttt{unlock()}\{\texttt{Lockset(}w \cdot \pi\texttt{)}\}}$$

(Unlock)

$$\frac{\Gamma \vdash v,\pi,w : \texttt{Object}, \texttt{lockset}, \texttt{Object}}{\Gamma; v \vdash \{\texttt{Lockset(}w \cdot \pi\texttt{)} * w.\texttt{inv}\}w.\texttt{unlock()}\{\texttt{Lockset(}\pi\texttt{)}\}}$$

K Verified Interfaces and Classes

Method subtyping. Let *MT* range over method types, as returned by the mtype function. Recall that these are slightly different from the method type representations in source code: *MT* makes the self-parameter explicit and existentially binds the result-parameter in the postcondition. We write $MT <: MT'$ when *MT* is a behavioral subtype of MT'. For a concrete definition of behavioral subtyping, we refer to [11], Section 3.2.

Predicate subtyping. Predicate type pt is a subtype of pt', if pt and pt' have the same name and pt's parameter signature "extends" pt''s parameter signature:

$$\frac{}{\texttt{pred}\ P\texttt{<}\bar{T}\ \bar{\alpha}, \bar{T}'\ \bar{\alpha}'\texttt{>} <: \texttt{pred}\ P\texttt{<}\bar{T}\ \bar{\alpha}\texttt{>}}$$

For qualified predicate types *fin pt*, we define subtyping as follows: *fin pt* $<:$ *fin*$'$ *pt*$'$ iff *fin*$' = \varepsilon$ and $pt <: pt'$.

Class extensions and interface implementations.

$$\begin{aligned}
\mathit{fin}\ \texttt{class}\ C\texttt{<}\bar{T}\ \bar{\alpha}\texttt{>}\ \texttt{ext}\ U\ \texttt{impl}\ \bar{V}\ \texttt{\{}\mathit{fd}\texttt{*}\ \mathit{pd}\texttt{*}\ \mathit{md}\texttt{*\}} &\Rightarrow \mathsf{methods}(C) \triangleq \mathsf{dom}(\mathit{md}\texttt{*}) \\
\texttt{interface}\ I\texttt{<}\bar{T}\ \bar{\alpha}\texttt{>}\ \texttt{ext}\ \bar{U}\ \texttt{\{}\mathit{pt}\texttt{*}\ \mathit{mt}\texttt{*\}} &\Rightarrow \mathsf{methods}(I) \triangleq \mathsf{dom}(\mathit{mt}\texttt{*}) \\
\mathit{fin}\ \texttt{class}\ C\texttt{<}\bar{T}\ \bar{\alpha}\texttt{>}\ \texttt{ext}\ U\ \texttt{impl}\ \bar{V}\ \texttt{\{}\mathit{fd}\texttt{*}\ \mathit{pd}\texttt{*}\ \mathit{md}\texttt{*\}} &\Rightarrow \mathsf{preds}(C) \triangleq \mathsf{dom}(\mathit{pd}\texttt{*}) \\
\texttt{interface}\ I\texttt{<}\bar{T}\ \bar{\alpha}\texttt{>}\ \texttt{ext}\ \bar{U}\ \texttt{\{}\mathit{pt}\texttt{*}\ \mathit{mt}\texttt{*\}} &\Rightarrow \mathsf{preds}(I) \triangleq \mathsf{dom}(\mathit{pt}\texttt{*}) \\
\mathit{fin}\ \texttt{class}\ C\texttt{<}\bar{T}\ \bar{\alpha}\texttt{>}\ \texttt{ext}\ U\ \texttt{impl}\ \bar{V}\ \texttt{\{}\mathit{fd}\texttt{*}\ \mathit{pd}\texttt{*}\ \mathit{md}\texttt{*\}} &\Rightarrow \mathsf{declared}(C) \triangleq \mathsf{dom}(\mathit{fd}\texttt{*})
\end{aligned}$$

In the following definitions, we conceive the partial functions mtype and ptype as total functions that map elements outside their domains to the special element undef. Furthermore, we extend the subtyping relation: $<: = \{(T,U) \mid T <: U\} \cup \{(\mathsf{undef}, \mathsf{undef})\}$.

$$C\texttt{<}\bar{T}\ \bar{\alpha}\texttt{>}\ \mathsf{ext}\ U \triangleq \begin{cases} U \text{ is a (parameterized) non-}\texttt{final}\text{ class} \\ f \in \mathsf{dom}(\mathsf{fld}(U)) \ \Rightarrow\ f \notin \mathsf{declared}(C) \\ (\forall m, mt)(\mathsf{mtype}(m,U) = MT \ \Rightarrow\ \bar{\alpha} : \bar{T} \vdash \mathsf{mtype}(m, C\texttt{<}\bar{\alpha}\texttt{>}) <: MT) \\ (\forall P, pt)(\mathsf{ptype}(P,U) = pt \ \Rightarrow\ \mathsf{ptype}(P, C\texttt{<}\bar{\alpha}\texttt{>}) <: pt) \end{cases}$$

$$I\texttt{<}\bar{T}\ \bar{\alpha}\texttt{>}\ \mathsf{type\text{-}extends}\ U \triangleq \begin{cases} U \text{ is a (parameterized) interface} \\ (\forall m, mt)(\mathsf{mtype}(m,U) = MT \ \Rightarrow\ \bar{\alpha} : \bar{T} \vdash \mathsf{mtype}(m, I\texttt{<}\bar{\alpha}\texttt{>}) <: MT) \\ (\forall P, pt)(\mathsf{ptype}(P,U) = pt \ \Rightarrow\ \mathsf{ptype}(P, I\texttt{<}\bar{\alpha}\texttt{>}) <: pt) \end{cases}$$

$$I\texttt{<}\bar{T}\ \bar{\alpha}\texttt{>}\ \mathsf{type\text{-}extends}\ \bar{U} \ \triangleq\ (\forall U \in \bar{U})(I\texttt{<}\bar{T}\ \bar{\alpha}\texttt{>}\ \mathsf{type\text{-}extends}\ U)$$

$$C\texttt{<}\bar{T}\ \bar{\alpha}\texttt{>}\ \mathsf{impl}\ U \triangleq \begin{cases} U \text{ is a (parameterized) interface} \\ (\forall m, mt)(\mathsf{mtype}(m,U) = MT \ \Rightarrow\ \mathsf{mtype}(m, C\texttt{<}\bar{\alpha}\texttt{>}) \neq \mathsf{undef}) \\ (\forall P, pt)(\mathsf{ptype}(P,U) = pt \ \Rightarrow\ \mathsf{ptype}(P, C\texttt{<}\bar{\alpha}\texttt{>}) \neq \mathsf{undef}) \\ (\forall m, mt)(\mathsf{mtype}(m,U) = MT \ \Rightarrow\ \bar{\alpha} : \bar{T} \vdash \mathsf{mtype}(m, C\texttt{<}\bar{\alpha}\texttt{>}) <: MT) \\ (\forall P, pt)(\mathsf{ptype}(P,U) = pt \ \Rightarrow\ \mathsf{ptype}(P, C\texttt{<}\bar{\alpha}\texttt{>}) <: pt) \end{cases}$$

$$C\texttt{<}\bar{T}\ \bar{\alpha}\texttt{>}\ \mathsf{impl}\ \bar{U} \ \triangleq\ (\forall U \in \bar{U})(C\texttt{<}\bar{T}\ \bar{\alpha}\texttt{>}\ \mathsf{impl}\ U)$$

Verified interfaces and classes.

Well-formed Predicate Types, $\Gamma \vdash \mathit{fin}\ pt : \diamond$, and Method Types, $\Gamma \vdash \mathit{fin}\ mt : \diamond$:

(Pred Type)
$$\frac{\Gamma \vdash \bar{T} : \diamond}{\Gamma \vdash \mathit{fin}\ \texttt{pred}\ P\texttt{<}\bar{T}\ \bar{\alpha}\texttt{>} : \diamond}$$

(Mth Type) $\quad m \in \{\texttt{run}, \texttt{start}\} \Rightarrow \Gamma(\texttt{this}) <: \texttt{Thread}$
$$\frac{\Gamma' = \Gamma, \bar{\alpha} : \bar{T}, \bar{\imath} : \bar{V} \quad \Gamma' \vdash \bar{T}, F, U, \bar{V} : \diamond \quad \Gamma'' = \Gamma', \texttt{result} : U \quad \Gamma'' \vdash G : \diamond}{\Gamma \vdash \mathit{fin}\ \texttt{<}\bar{T}\ \bar{\alpha}\texttt{>}\,\texttt{req}\,F\texttt{;}\,\texttt{ens}\,G\texttt{;}\ U\ m\texttt{(}\bar{V}\ \bar{\imath}\texttt{)} : \diamond}$$

Verified Interfaces, $int : \diamond$**:**

$$\text{(Int)}\quad \frac{\bar{\alpha} : \bar{T} \vdash \bar{T}, \bar{U}, pt* : \diamond \quad \bar{\alpha} : \bar{T}, \texttt{this} : I\texttt{<}\bar{\alpha}\texttt{>} \vdash mt* : \diamond}{\texttt{interface}\ I\texttt{<}\bar{T}\ \bar{\alpha}\texttt{>}\ \texttt{ext}\ \bar{U}\ \texttt{\{}pt*\ mt*\texttt{\}} : \diamond} \quad I\texttt{<}\bar{T}\ \bar{\alpha}\texttt{>}\ \mathsf{type\text{-}extends}\ \bar{U} \quad \texttt{init} \notin \mathsf{dom}(pt*)$$

Recall that $\mathsf{cfv}(c)$ is the set of variables that occur freely in an object creation command in c. In the rule (Mth) below, we prohibit object creation commands to contain logical method parameters. This is needed because our operational semantics does not keep track of logical method parameters (while it does keep track of class parameters).

Verified Classes, $cl : \diamond$**:**

$$\text{(Cls)}\quad \frac{\bar{\alpha} : \bar{T} \vdash \bar{T}, U, \bar{V} : \diamond \qquad \bar{\alpha} : \bar{T} \vdash pd* : \diamond\ \mathsf{in}\ C\texttt{<}\bar{\alpha}\texttt{>} \qquad \bar{\alpha} : \bar{T}, \texttt{this} : C\texttt{<}\bar{\alpha}\texttt{>} \vdash fd*, md* : \diamond}{fin\ \texttt{class}\ C\texttt{<}\bar{T}\ \bar{\alpha}\texttt{>}\ \texttt{ext}\ U\ \texttt{impl}\ \bar{V}\ \texttt{\{}fd*\ pd*\ md*\texttt{\}} : \diamond} \quad C\texttt{<}\bar{T}\ \bar{\alpha}\texttt{>}\ \mathsf{impl}\ \bar{V} \quad \texttt{init} \notin \mathsf{dom}(pd*)$$

$$\text{(Fld)}\quad \frac{\Gamma \vdash T : \diamond}{\Gamma \vdash T\ f : \diamond} \qquad \text{(Pred)}\quad \frac{\Gamma \vdash fin\ \texttt{pred}\ P\texttt{<}\bar{T}\ \bar{\alpha}\texttt{>} : \diamond \quad \Gamma, \texttt{this} : U, \bar{\alpha} : \bar{T} \vdash F : \diamond}{\Gamma \vdash fin\ \texttt{pred}\ P\texttt{<}\bar{T}\ \bar{\alpha}\texttt{>=}F : \diamond\ \mathsf{in}\ U}$$

$$\text{(Mth)}\quad \frac{\Gamma' = \Gamma, \bar{\alpha} : \bar{T}, \bar{\imath} : \bar{V} \quad \Gamma'; \texttt{this} \vdash \{F * \texttt{this} \neq \texttt{null} * F'\} c : U \{(\texttt{ex}\ U\ \texttt{result})(G)\}}{\Gamma \vdash fin\ \texttt{<}\bar{T}\ \bar{\alpha}\texttt{>req}\ F\texttt{;ens}\ G\texttt{;}\ U\ m\texttt{(}\bar{V}\ \bar{\imath}\texttt{)\{}c\texttt{\}} : \diamond} \quad \begin{array}{c} m \neq \texttt{run} \Rightarrow F' = \texttt{true} \quad m = \texttt{run} \Rightarrow F' = \texttt{Lockset(nil)} \\ \Gamma \vdash fin\ \texttt{<}\bar{T}\ \bar{\alpha}\texttt{>req}\ F\texttt{;ens}\ G\texttt{;}\ U\ m\texttt{(}\bar{V}\ \bar{\imath}\texttt{)} : \diamond \quad \mathsf{cfv}(c) \cap \bar{\alpha} = \emptyset \end{array}$$

L Operational Semantics

Recall the definition of runtime structures from Section 6.1.

Auxiliary syntax for wait/notify. To represent states in which threads are waiting to be notified, we could associate each object with a set of waiting threads (the "wait set"). However, we prefer to avoid introducing yet another runtime structure, and therefore represent waiting states syntactically:

$$hc ::= \ldots \mid o\texttt{.waiting(}n\texttt{)} \mid o\texttt{.resume(}n\texttt{)}$$

Restriction: These clauses must not occur in source programs.

Here are the intuitive semantics of these head commands:

- o`.waiting(`n`)`: If thread p's head command is o`.waiting(`n`)`, then p is waiting to be notified to resume competition for o's lock at reentrancy level n.
- o`.resume(`n`)`: If thread p's head command is o`.resume(`n`)`, then p has been notified to resume competition for o's lock at reentrancy level n, and is now competing for this lock.

Our preservation theorem says that all states that are reachable from verified programs are "well-verified". We have to extend our definition of "well-verified" states to deal with the new auxiliary syntax. To this end, we add the following Hoare rules:

(Waiting)

$$\frac{\Gamma \vdash r, \pi, o : \texttt{Object}, \texttt{lockset}, \texttt{Object}}{\Gamma; r \vdash \{\texttt{Lockset(}\pi\texttt{)} * o.\texttt{initialized}\} o.\texttt{waiting(}n\texttt{)} \{\texttt{Lockset(}\pi\texttt{)} * o.\texttt{initialized}\}}$$

(Resume)

$$\frac{\Gamma \vdash o, \pi : \texttt{Object}, \texttt{lockset}}{\Gamma; r \vdash \{\texttt{Lockset(}\pi\texttt{)} * o.\texttt{initialized}\} o.\texttt{resume(}n\texttt{)} \{\texttt{Lockset(}o^n \cdot \pi\texttt{)} * o.\texttt{inv}\}}$$

In (Resume), o^n denotes the multiset with n occurrences of o. More precisely: $o^0 = \texttt{nil}$, and $o^n = o \cdot o^{n-1}$ if $n \geq 1$. Of course, the rules (Waiting) and (Resume) are never used in source code verification, because source programs do not contain the auxiliary syntax.

Auxiliary syntax for method call/return. In order to model method call/return, we could introduce proper stacks. However, we prefer to avoid introducing another runtime structure, and therefore represent method calls by a derived form that "flattens" the stack. To this end, we first define another piece of auxiliary syntax:

$$c ::= \ldots \mid \ell\texttt{=return(}v\texttt{);}\, c$$

Restriction: This clause must not occur in source programs.

We associate this auxiliary syntax with the following Hoare rule:

(Return)

$$\frac{\Gamma \vdash v : T \quad \Gamma; o; F \vdash G[v/\alpha] \quad T <: U \quad \Gamma, \ell : U; p \vdash \{\texttt{(ex } T\ \alpha\texttt{)(}\alpha\texttt{ == }\ell\texttt{ * }G\texttt{)}\} c : V\{H\}}{\Gamma, \ell : U; o \vdash \{F\}\ell\texttt{=return(}v\texttt{);}\, c : V\{H\}}$$

Note that the rule reflects that the receiver changes at method return: the receiver is o before return, and p after return.

Now we define a derived form, $\ell \leftarrow c; c'$, which assigns the result of a computation c to variable ℓ. In our applications of this derived form, c is always a source program command and we can therefore assume that c does not contain `return`-commands.

$$\begin{array}{rll}
\ell \leftarrow v; c & \triangleq & \ell\texttt{=return(}v\texttt{);}\, c \\
\ell \leftarrow (U\ \ell'; c); c' & \triangleq & U\ \ell'; \ell \leftarrow c; c' \qquad \text{if } \ell' \notin \mathsf{fv}(c'),\ \ell' \neq \ell \\
\ell \leftarrow (\texttt{final}\ U\ \iota\texttt{=}\ell'; c); c' & \triangleq & \texttt{final}\ U\ \iota\texttt{=}\ell'; \ell \leftarrow c; c' \qquad \text{if } \iota \notin \mathsf{fv}(c') \\
\ell \leftarrow (hc; c); c' & \triangleq & hc; \ell \leftarrow c; c'
\end{array}$$

We define sequential composition of commands:

$$c; c' \ \triangleq\ \texttt{void}\ \ell; \ell \leftarrow c; c' \quad \text{where } \ell \notin \mathsf{fv}(c, c')$$

Default values and initial object stores. We define a function that maps types to their default values:

$$\mathsf{df} : \mathsf{Type} \to \mathsf{ClVal}$$

$$\mathsf{df}(C\texttt{<}\pi\texttt{>}) \triangleq \texttt{null} \quad \mathsf{df}(\texttt{void}) \triangleq \texttt{null} \quad \mathsf{df}(\texttt{int}) \triangleq 0 \quad \mathsf{df}(\texttt{bool}) \triangleq \texttt{false}$$

We define a function that maps object types to their initial object stores:

$$\mathsf{init} : \mathsf{Type} \rightharpoonup \mathsf{ObjStore} \quad \mathsf{init}(t\texttt{<}\bar{\pi}\texttt{>})(f) = \mathsf{df}(T) \text{ iff } (T\ f) \in \mathsf{fld}(C\texttt{<}\pi\texttt{>})$$

Small-step reduction. The state reduction relation $\rightarrow_{ct}$ is given with respect to a class table ct. We usually omit the subscript ct. In the reduction rules, we use the following abbreviation for field updates: $h[o.f \mapsto v] = h[o \mapsto (h(o)_1, h(o)_2[f \mapsto v])]$

State Reductions, $st \rightarrow_{ct} st'$:

(Red Dcl) $\ell \notin \mathsf{dom}(s) \quad s' = s[\ell \mapsto \mathsf{df}(T)]$
$\langle h, ts \mid p \text{ is } (s \text{ in } T\ \ell; c)\rangle \rightarrow \langle h, ts \mid p \text{ is } (s' \text{ in } c)\rangle$

(Red Fin Dcl) $s(\ell) = v \quad c' = c[v/\iota]$
$\langle h, ts \mid p \text{ is } (s \text{ in } \texttt{final}\ T\ \iota\texttt{=}\ell; c)\rangle \rightarrow \langle h, ts \mid p \text{ is } (s \text{ in } c')\rangle$

(Red Var Set) $s' = s[\ell \mapsto v]$
$\langle h, ts \mid p \text{ is } (s \text{ in } \ell\texttt{=}v; c)\rangle \rightarrow \langle h, ts \mid p \text{ is } (s' \text{ in } c)\rangle$

(Red Op) $\mathsf{arity}(op) = |\vec{v}| \quad [\![op]\!]^h(\vec{v}) = w \quad s' = s[\ell \mapsto w]$
$\langle h, ts \mid p \text{ is } (s \text{ in } \ell\texttt{=}op(\vec{v}); c)\rangle \rightarrow \langle h, ts \mid p \text{ is } (s' \text{ in } c)\rangle$

(Red Get) $s' = s[\ell \mapsto h(o)_2(f)]$
$\langle h, ts \mid p \text{ is } (s \text{ in } \ell\texttt{=}o.f; c)\rangle \rightarrow \langle h, ts \mid p \text{ is } (s' \text{ in } c)\rangle$

(Red Set) $h' = h[o.f \mapsto v]$
$\langle h, ts \mid p \text{ is } (s \text{ in } o.f\texttt{=}v; c)\rangle \rightarrow \langle h', ts \mid p \text{ is } (s \text{ in } c)\rangle$

(Red New) $o \notin \mathsf{dom}(h) \quad h' = h[o \mapsto (C\texttt{<}\vec{\pi}\texttt{>}, \mathsf{init}(C\texttt{<}\vec{\pi}\texttt{>}))] \quad s' = s[\ell \mapsto o] \quad l' = l[o \mapsto \mathsf{free}]$
$\langle h, l, ts \mid p \text{ is } (s \text{ in } \ell\texttt{=new}\ C\texttt{<}\vec{\pi}\texttt{>}; c)\rangle \rightarrow \langle h', l', ts \mid p \text{ is } (s' \text{ in } c)\rangle$

(Red Call) $m \notin \{\texttt{start}, \texttt{wait}, \texttt{notify}\}$
$h(o)_1 = C\texttt{<}\vec{\pi}'\texttt{>} \quad \mathsf{mbody}(m, C\texttt{<}\vec{\pi}'\texttt{>}) = (\iota_0, \vec{\iota}).c_m \quad c' = c_m[o/\iota_0, \vec{v}/\vec{\iota}]$
$\langle h, ts \mid p \text{ is } (s \text{ in } \ell\texttt{=}o.m(\vec{v}); c)\rangle \rightarrow \langle h, ts \mid p \text{ is } (s \text{ in } \ell \leftarrow c'; c)\rangle$

(Red Return)
$\langle h, ts \mid p \text{ is } (s \text{ in } \ell\texttt{=return(}v\texttt{)}; c)\rangle \rightarrow \langle h, ts \mid p \text{ is } (s \text{ in } \ell\texttt{=}v; c)\rangle$

(Red If True)
$\langle h, ts \mid p \text{ is } (s \text{ in } \texttt{if (true)\{}c\texttt{\}else\{}c'\texttt{\}}; c'')\rangle \rightarrow \langle h, ts \mid p \text{ is } (s \text{ in } c; c'')\rangle$

(Red If False)
$\langle h, ts \mid p \text{ is } (s \text{ in } \texttt{if (false)\{}c\texttt{\}else\{}c'\texttt{\}}; c'')\rangle \rightarrow \langle h, ts \mid p \text{ is } (s \text{ in } c'; c'')\rangle$

(Red Start) $h(o)_1 = C\texttt{<}\vec{\pi}\texttt{>} \quad o \notin \mathsf{dom}(ts), \{p\} \quad \mathsf{mbody}(\texttt{run}, C\texttt{<}\vec{\pi}\texttt{>}) = (\iota).c_r \quad c_o = c_r[o/\iota]$
$\langle h, ts \mid p \text{ is } (s \text{ in } \ell\texttt{=}o\texttt{.start();}\ c)\rangle \rightarrow \langle h, ts \mid p \text{ is } (s \text{ in } c) \mid o \text{ is } (\emptyset \text{ in } c_o)\rangle$

(Red Lock) $(l(o) = \mathsf{free},\ l' = l[o \mapsto (1,p)])$ or $(l(o) = (n,p),\ l' = l[o \mapsto (n+1,p)])$
$\langle h, l, ts \mid p \text{ is } (s \text{ in } o\texttt{.lock();}\ c)\rangle \rightarrow \langle h, l', ts \mid p \text{ is } (s \text{ in } c)\rangle$

(Red Unlock) $l(o) = (n,p) \quad n = 1 \Rightarrow l' = l[o \mapsto \mathsf{free}] \quad n > 1 \Rightarrow l' = l[o \mapsto (n-1,p)]$
$\langle h, l, ts \mid p \text{ is } (s \text{ in } o\texttt{.unlock();}\ c)\rangle \rightarrow \langle h, l', ts \mid p \text{ is } (s \text{ in } c)\rangle$

(Red Wait) $l(o) = (n,p) \quad l' = l[o \mapsto \mathsf{free}]$
$\langle h, l, ts \mid p \text{ is } (s \text{ in } \ell\texttt{=}o\texttt{.wait();}\ c)\rangle \rightarrow \langle h, l', ts \mid p \text{ is } (s \text{ in } o\texttt{.waiting(}n\texttt{);}\ o\texttt{.resume(}n\texttt{);}\ c)\rangle$

(Red Notify) $l(o) = (n,p)$
$\langle h, l, ts \mid p \text{ is } (s \text{ in } \ell\texttt{=}o\texttt{.notify();}\ c) \mid q \text{ is } (s' \text{ in } o\texttt{.waiting(}n'\texttt{);}\ c')\rangle \rightarrow$
$\langle h, l, ts \mid p \text{ is } (s \text{ in } c) \mid q \text{ is } (s' \text{ in } c')\rangle$

(Red Skip Notify) $l(o) = (n,p)$
$\langle h, l, ts \mid p \text{ is } (s \text{ in } \ell\texttt{=}o\texttt{.notify();}\ c)\rangle \rightarrow \langle h, l, ts \mid p \text{ is } (s \text{ in } c)\rangle$

(Red Resume) $l(o) = \mathsf{free} \quad l' = l[o \mapsto (n,p)]$
$\langle h, l, ts \mid p \text{ is } (s \text{ in } o\texttt{.resume(}n\texttt{);}\ c)\rangle \rightarrow \langle h, l', ts \mid p \text{ is } (s \text{ in } c)\rangle$

(Red No Op)
$\langle h, ts \mid p \text{ is } (s \text{ in } sc; c)\rangle \rightarrow \langle h, ts \mid p \text{ is } (s \text{ in } c)\rangle$

M Predicate Environments

The predicates that are declared in the class table define a function from predicate symbols to relations. This function is called a *predicate environment*.

Predicate domains. What is the domain of these relations? Roughly, the domain consists of resources (including the heap) and tuples of specification values (representing class parameters and predicate parameters). We now define predicate domains precisely. First, let $\mathsf{SpecVals}$ be the set of all tuples of specification values:

$$\mathsf{SpecVals} \triangleq \bigcup_{n\geq 0} \mathsf{SpecVal}^n$$

Let $\mathsf{Pred}(ct)$ be the set of all qualified predicates $P@C$ that are defined in class table ct:

$$\mathsf{Pred}(ct) \triangleq \{\ P@C \mid C \in \mathsf{dom}(ct) \text{ and } P \text{ is defined in } C\ \}$$

For $P@C$ in $\mathsf{Pred}(ct)$, its domain $\mathsf{Dom}(P@C)$ is defined as the subset of $\mathsf{SpecVals} \times \mathsf{Resources} \times \mathsf{ObjId} \times \mathsf{SpecVals}$ that consists of all tuples $(\bar{\pi}, \mathscr{R}, r, \bar{\pi}')$ that satisfy the following conditions:

(a) $\mathsf{fst} \circ \mathscr{R}_{\mathsf{hp}} \vdash r : C\texttt{<}\bar{\pi}\texttt{>}$.
(b) $\mathsf{ptype}(P, C\texttt{<}\bar{\pi}\texttt{>}) = \mathit{fin}\ \texttt{pred}\ P\texttt{<}\bar{T}\ \bar{\alpha}\texttt{>}$ and $\mathsf{fst} \circ \mathscr{R}_{\mathsf{hp}} \vdash \bar{\pi}' : \bar{T}$ for some $\mathit{fin}, \mathit{pmod}, \bar{T}, \bar{\alpha}$.

Intuitively, if $(\bar{\pi}, \mathscr{R}, r, \bar{\pi}') \in \mathsf{Dom}(P@C)$, then $\bar{\pi}$ represents the class parameters of r's dynamic class C, r represents the predicate receiver, and $\bar{\pi}'$ represents P's actual predicate parameters.

Predicate environments. We choose to represent relations as functions into the two-element set: Let $\mathbb{2}$ be the two-element set $\{0, 1\}$ equipped with the usual order (i.e., $0 \leq 1$). A *predicate environment* is a function of type $\prod \kappa \in \mathsf{Pred}(ct).\mathsf{Dom}(\kappa) \rightarrow \mathbb{2}$ such that the following axioms hold:

(a) If $(\bar{\pi}, \mathscr{R}, r, \bar{\pi}'), (\bar{\pi}, \mathscr{R}', r, \bar{\pi}') \in \mathsf{Dom}(\kappa)$ and $\mathscr{R} \leq \mathscr{R}'$,
then $\mathscr{E}(\kappa)(\bar{\pi}, \mathscr{R}, r, \bar{\pi}') \leq \mathscr{E}(\kappa)(\bar{\pi}, \mathscr{R}', r, \bar{\pi}')$.
(b) If $(\bar{\pi}, \mathscr{R}, r, \bar{\pi}'), (\bar{\pi}, \mathscr{R}', r, \bar{\pi}') \in \mathsf{Dom}(\kappa)$,
$(\mathscr{R}_{\mathsf{hp}}, \mathscr{R}_{\mathsf{perm}}, \mathscr{R}_{\mathsf{lock}}, \mathscr{R}_{\mathsf{fresh}}) = (\mathscr{R}'_{\mathsf{hp}}, \mathscr{R}'_{\mathsf{perm}}, \mathscr{R}'_{\mathsf{lock}}, \mathscr{R}'_{\mathsf{fresh}})$ and $\mathscr{R}_{\mathsf{init}} \subseteq \mathscr{R}'_{\mathsf{init}}$,
then $\mathscr{E}(\kappa)(\bar{\pi}, \mathscr{R}, r, \bar{\pi}') \leq \mathscr{E}(\kappa)(\bar{\pi}, \mathscr{R}', r, \bar{\pi}')$.

Axiom (a) says that predicates are monotone in the resources[7]: if a predicate is satisfied in resource $\mathscr{R}$, then it is also satisfied in all larger resources $\mathscr{R}'$. This axiom is natural for a language with garbage collection. Axiom (b) says that predicates are monotone with respect to initialized sets.

[7] The resource order is defined as usual: $\mathscr{R} \leq \mathscr{R}'$ iff $(\exists \mathscr{R}'')(\mathscr{R}' = \mathscr{R} * \mathscr{R}'')$.

The class table's predicate environment. The class table ct defines a predicate environment that maps each predicate in ct to its definition. Technically, this predicate environment is defined as the least fixed point of the endofunction $\mathscr{F}_{ct}$ on predicate environments. The definition of $\mathscr{F}_{ct}$ refers to the Kripke resource semantics $\models$, as defined in Appendix N.

$$\frac{\begin{array}{l}\mathsf{pbody}(r.P\texttt{<}\bar{\pi}'\texttt{>},C\texttt{<}\bar{\pi}\texttt{>}) = F\ \texttt{ext}\ D\texttt{<}\bar{\pi}''\texttt{>} \\ C \neq \texttt{Object} \text{ and } \mathsf{arity}(P,D) = n \ \Rightarrow\ F' = r.P\texttt{@}D\texttt{<}\bar{\pi}'_{\mathsf{to}\,n}\texttt{>} \\ C = \texttt{Object} \text{ or } P \text{ is rooted in } C \ \Rightarrow\ F' = \texttt{true}\end{array}}{\mathscr{F}_{ct}(\mathscr{E})(P\texttt{@}C)(\bar{\pi},\mathscr{R},r,\bar{\pi}') = \begin{cases}1 & \text{if } \mathsf{fst}\circ\mathscr{R}_{\mathsf{hp}} \vdash \mathscr{E};\mathscr{R};\emptyset \models F * F' \\ 0 & \text{otherwise}\end{cases}}$$

In this definition, $\bar{\pi}'_{\mathsf{to}\,n}$ denotes the tuple consisting of the first n entries of $\bar{\pi}'$ (which may have more than n entries due to arity extension in subclasses).

Lemma 3 (Well-Typedness of $\mathscr{F}_{ct}$). *If $\mathscr{E}$ is a pre-environment over X, then so is $\mathscr{F}_{ct}(\mathscr{E})$.*

Proof. We need to show that $\mathscr{F}_{ct}(\mathscr{E})$ satisfies the axioms for predicate environments. This is a consequence of two lemmas that we show later: axiom (a) is a consequence of Lemma 30, and axiom (b) is a consequence of Lemma 34. □

Theorem 1 (Existence of Fixed Points). *If $ct : \diamond$, then there exists a predicate environment $\mathscr{E}$ such that $\mathscr{F}_{ct}(\mathscr{E}) = \mathscr{E}$.*

Proof. This is a consequence of a fixed point theorem (as stated and proven in [12], Theorem 8). The fixed point theorem makes use of the fact that, by syntactic restriction, cyclic predicate dependencies must be positive (i.e., dependency cycles must not contain negative dependencies). This restriction is certainly satisfied when predicate occurrences in negative positions are forbidden altogether, which is the restriction that Parkinson and Bierman impose [22]. □

N Kripke Resource Semantics

Let $(\Gamma \vdash \mathscr{R} : \diamond)$ whenever $\Gamma \vdash \mathscr{R}_{\mathsf{hp}} : \diamond$, $\mathscr{P}(o,f) > 0$ implies $o \in \mathsf{dom}(\Gamma)$, $\mathsf{supp}(\mathscr{R}_{\mathsf{lock}}) \subseteq \mathsf{dom}(\Gamma)$, $\mathsf{dom}(\mathscr{R}_{\mathsf{fresh}}) \subseteq \mathsf{dom}(\Gamma)$, and $\mathsf{dom}(\mathscr{R}_{\mathsf{init}}) \subseteq \mathsf{dom}(\Gamma)$. Let $(\Gamma \vdash \mathscr{E},\mathscr{R},s,F : \diamond)$ whenever $\Gamma \vdash \mathscr{R} : \diamond$, $\Gamma \vdash s : \diamond$, and $\Gamma \vdash F : \diamond$.

The relation $(\Gamma \vdash \mathscr{E};\mathscr{R};s \models F)$ is the unique subset of $(\Gamma \vdash \mathscr{E},\mathscr{R},s,F : \diamond)$ that satisfies the following clauses:

$$
\begin{array}{rll}
\Gamma \vdash (h,\mathscr{P},\mathscr{L},\mathscr{F},\mathscr{I});s \models e & \text{iff} & [\![e]\!]^h_s = \texttt{true} \\
\Gamma \vdash \mathscr{E};(h,\mathscr{P},\mathscr{L},\mathscr{F},\mathscr{I});s \models \texttt{PointsTo}(e.f,\pi,e') & \text{iff} & \begin{cases} [\![e]\!]^h_s = o, h(o)_2(f) = [\![e']\!]^h_s \text{ and} \\ [\![\pi]\!] \leq \mathscr{P}_1(o,f) \end{cases} \\
\Gamma \vdash \mathscr{E};(h,\mathscr{P},\mathscr{L},\mathscr{F},\mathscr{I});s \models \texttt{Lockset}(\pi) & \text{iff} & \mathscr{L}(o) = [\![\pi]\!] \text{ for some } o \\
\Gamma \vdash \mathscr{E};(h,\mathscr{P},\mathscr{L},\mathscr{F},\mathscr{I});s \models \pi \texttt{ contains } e & \text{iff} & [\![e]\!]^h_s \in [\![\pi]\!] \\
\Gamma \vdash \mathscr{E};(h,\mathscr{P},\mathscr{L},\mathscr{F},\mathscr{I});s \models e.\texttt{fresh} & \text{iff} & [\![e]\!]^h_s \in \mathscr{F} \\
\Gamma \vdash \mathscr{E};(h,\mathscr{P},\mathscr{L},\mathscr{F},\mathscr{I});s \models e.\texttt{initialized} & \text{iff} & [\![e]\!]^h_s \in \mathscr{I} \\
\Gamma \vdash \mathscr{E};\mathscr{R};s \models \texttt{null.}\kappa\texttt{<}\bar{\pi}\texttt{>} & \text{iff} & \text{true} \\
\Gamma \vdash \mathscr{E};\mathscr{R};s \models o.P\texttt{@}C\texttt{<}\bar{\pi}\texttt{>} & \text{iff} & \begin{cases} \mathscr{R}_{\mathsf{hp}}(o)_1 <: C\texttt{<}\bar{\pi}'\texttt{>} \text{ and} \\ \mathscr{E}(P\texttt{@}C)(\bar{\pi}',\mathscr{R},o,\bar{\pi}) = 1 \end{cases} \\
\Gamma \vdash \mathscr{E};\mathscr{R};s \models o.P\texttt{<}\bar{\pi}\texttt{>} & \text{iff} & \begin{cases} (\exists \bar{\pi}'')(\mathscr{R}_{\mathsf{hp}}(o)_1 = C\texttt{<}\bar{\pi}'\texttt{>} \text{ and} \\ \mathscr{E}(P\texttt{@}C)(\bar{\pi}',\mathscr{R},o,(\bar{\pi},\bar{\pi}'')) = 1) \end{cases} \\
\Gamma \vdash \mathscr{E};\mathscr{R};s \models F * G & \text{iff} & \begin{cases} (\exists \mathscr{R}_1,\mathscr{R}_2)(\mathscr{R} = \mathscr{R}_1 * \mathscr{R}_2, \\ \Gamma \vdash \mathscr{E};\mathscr{R}_1;s \models F \text{ and } \Gamma \vdash \mathscr{E};\mathscr{R}_2;s \models G) \end{cases} \\
\Gamma \vdash \mathscr{E};\mathscr{R};s \models F \mathrel{-\!\!*} G & \text{iff} & \begin{cases} (\forall \Gamma' \supseteq_{\mathsf{hp}} \Gamma, \mathscr{R}')(\\ \mathscr{R}\#\mathscr{R}' \text{ and } \Gamma' \vdash \mathscr{E};\mathscr{R}';s \models F \\ \Rightarrow\ \Gamma' \vdash \mathscr{E};\mathscr{R} * \mathscr{R}';s \models G\ ) \end{cases} \\
\Gamma \vdash \mathscr{E};\mathscr{R};s \models F \texttt{ \& } G & \text{iff} & \Gamma \vdash \mathscr{E};\mathscr{R};s \models F \text{ and } \Gamma \vdash \mathscr{E};\mathscr{R};s \models G \\
\Gamma \vdash \mathscr{E};\mathscr{R};s \models F \texttt{ | } G & \text{iff} & \Gamma \vdash \mathscr{E};\mathscr{R};s \models F \text{ or } \Gamma \vdash \mathscr{E};\mathscr{R};s \models G \\
\Gamma \vdash \mathscr{E};\mathscr{R};s \models \texttt{(ex } T\ \alpha\texttt{)(}F\texttt{)} & \text{iff} & \begin{cases} (\exists \pi)(\ \Gamma_{\mathsf{hp}} \vdash \pi : T \text{ and} \\ \Gamma \vdash \mathscr{E};\mathscr{R};s \models F[\pi/\alpha]\ ) \end{cases} \\
\Gamma \vdash \mathscr{E};\mathscr{R};s \models \texttt{(fa } T\ \alpha\texttt{)(}F\texttt{)} & \text{iff} & \begin{cases} (\forall \Gamma' \supseteq_{\mathsf{hp}} \Gamma, \mathscr{R}' \geq \mathscr{R}, \pi)(\\ \Gamma'_{\mathsf{hp}} \vdash \mathscr{R}'_{\mathsf{hp}} : \diamond \text{ and } \Gamma'_{\mathsf{hp}} \vdash \pi : T \\ \Rightarrow\ \Gamma' \vdash \mathscr{E};\mathscr{R}';s \models F[\pi/\alpha]\ ) \end{cases}
\end{array}
$$

Soundness of the proof theory. We define semantic entailment $\Gamma \vdash \mathscr{E};\bar{F} \models G$:

$$
\begin{array}{rll}
\Gamma \vdash \mathscr{E};\mathscr{R};s \models F_1,\ldots,F_n & \text{iff} & \Gamma \vdash \mathscr{E};\mathscr{R};s \models F_1 * \cdots * F_n \\
\Gamma \vdash \mathscr{E};\bar{F} \models G & \text{iff} & (\forall \Gamma,\mathscr{R},s)(\Gamma \vdash \mathscr{E};\mathscr{R};s \models \bar{F} \ \Rightarrow\ \Gamma \vdash \mathscr{E};\mathscr{R};s \models G)
\end{array}
$$

Now, we can show soundness of the proof theory:

Theorem 2 (Soundness of Logical Consequence). *If $\mathscr{F}_{ct}(\mathscr{E}) = \mathscr{E}$ and $(\Gamma;o;\bar{F} \vdash G)$, then $(\Gamma \vdash \mathscr{E};\bar{F} \models G)$.*

Proof. The proof is by induction on $(\Gamma;o;\bar{F} \vdash G)$'s proof tree, using some standard lemmas from Appendix O. Proof details for most proof cases can be found in the proof of Lemma 81 in [12]. In addition, we have to prove soundness of the multiset axioms and the copyability axioms, which were not part of [12]. These additional proof cases are routine. □

O Properties

In this section, we collect standard lemmas that the system is designed to satisfy.

O.1 Properties of the Typing Judgments

Lemma 4 (Good Environments). *Let $\mathscr{J}$ range over right-hand sides of the forms $T : \diamond$, $v : T$, $\pi : T$, $e : T$, $F : \diamond$, $s : \diamond$, $obj : \diamond$ and $h : \diamond$. If $(\Gamma \vdash \mathscr{J})$, then $(\Gamma \vdash \diamond)$.*

Proof. By induction on the derivation of $(\Gamma \vdash \mathscr{J})$. □

Lemma 5 (Weakening). *Let $\mathscr{J}$ range over right-hand sides of the forms $T : \diamond$, $v : T$, $\pi : T$, $e : T$, $F : \diamond$, $s : \diamond$, $obj : \diamond$ and $h : \diamond$. If $(\Gamma \vdash \mathscr{J})$, $\Gamma \subseteq \Gamma'$ and $(\Gamma' \vdash \diamond)$, then $(\Gamma' \vdash \mathscr{J})$.*

Proof. By induction on the derivation of $(\Gamma \vdash \mathscr{J})$. □

Lemma 6 (Strengthening). *Let $\mathscr{J}$ range over right-hand sides of the forms $\diamond$, $U : \diamond$, $v : U$, $\pi : U$, $e : U$, $F : \diamond$ and $obj : \diamond$. If $(\Gamma, x : T \vdash \mathscr{J})$ and $x \notin \mathsf{fv}(\Gamma, \mathscr{J})$, then $(\Gamma \vdash \mathscr{J})$.*

Proof. By induction on the derivation of $(\Gamma, x : T \vdash \mathscr{J})$. □

Lemma 7 (Substitutivity and Inverse Substitutivity for Subtyping).

(a) *If $T <: U$, then $T[\sigma] <: U[\sigma]$.*
(b) *If $T[\sigma] <: U$, then $U = U'[\sigma]$ for some U'.*
(c) *If $T[\sigma] <: U[\sigma]$, then $T <: U$.*

Proof. All three parts by induction on the derivation of the subtyping judgment. The proof of part (c) uses part (b) to deal with the transitivity rule. □

Lemma 8 (Substitutivity). *Let $\mathscr{J}$ range over right-hand-sides of the forms $T : \diamond$, $v : U$, $\pi : U$, $e : U$ and $F : \diamond$.*

(a) *If $(\Gamma[\bar{\pi}/\bar{x}] \vdash \bar{\pi} : \bar{T}[\bar{\pi}/\bar{x}])$ and $(\Gamma, \bar{x} : \bar{T} \vdash \mathscr{J})$, then $(\Gamma[\bar{\pi}/\bar{x}] \vdash \mathscr{J}[\bar{\pi}/\bar{x}])$.*
(b) *If $(\Gamma \vdash \bar{e} : \bar{T})$ and $(\Gamma, \bar{\ell} : \bar{T} \vdash \mathscr{J})$, then $(\Gamma \vdash \mathscr{J}[\bar{e}/\bar{\ell}])$.*
(c) *If $(\Gamma \vdash \sigma : \diamond)$ and $(\Gamma \vdash \mathscr{J})$, then $(\Gamma_{\mathsf{hp}} \vdash \mathscr{J}[\sigma])$.*[8]

Proof. Part (a) by induction on $(\Gamma, \bar{x} : \bar{T} \vdash \mathscr{J})$. Part (b) by induction on $(\Gamma, \bar{\ell} : \bar{T} \vdash \mathscr{J})$. Part (c) follows from part (a) in the following way: Suppose $(\Gamma \vdash \sigma : \diamond)$ and $(\Gamma \vdash \mathscr{J})$. Let $\bar{x} = \mathsf{dom}(\Gamma) \cap \mathsf{Var}$. If $\bar{x} = \emptyset$, then $\Gamma_{\mathsf{hp}} = \Gamma$, $\sigma = \emptyset$ and $\mathscr{J}[\sigma] = \mathscr{J}$. $(\Gamma_{\mathsf{hp}} \vdash \mathscr{J}[\sigma])$ trivially follows. So suppose $\bar{x} \neq \emptyset$. Then $(\Gamma_{\mathsf{hp}} \vdash \sigma(\bar{x}) : \Gamma(\bar{x})[\sigma])$, by definition of $(\Gamma \vdash \sigma : \diamond)$. In particular, it follows that $(\Gamma_{\mathsf{hp}} \vdash \diamond)$ (by Lemma 4) and therefore $\mathsf{fv}(\Gamma_{\mathsf{hp}}) = \emptyset$. Therefore, $\Gamma_{\mathsf{hp}}[\sigma] = \Gamma_{\mathsf{hp}}$ and, thus, $(\Gamma_{\mathsf{hp}}[\sigma] \vdash \sigma(\bar{x}) : \Gamma(\bar{x})[\sigma])$. Now, we can apply part (a) to obtain $(\Gamma_{\mathsf{hp}} = \Gamma_{\mathsf{hp}}[\sigma] \vdash \mathscr{J}[\sigma])$. □

Lemma 9 (Inverse Substitutivity for Values). *If $(\Gamma \vdash \sigma : \Gamma')$ and $(\Gamma[\sigma] \vdash v : T[\sigma])$, then $(\Gamma, \Gamma' \vdash v : T)$.*

Proof. In case v is an integer, boolean or `null`, this is obvious. So suppose that v is an object identifier or a read-only variable. Then $v \in \mathsf{dom}(\Gamma)$ and $\Gamma[\sigma](v) <: T[\sigma]$. But then $\Gamma(v) <: T$, by Lemma 7. But then $(\Gamma, \Gamma' \vdash v : T)$. □

Lemma 10. *If $(\Gamma \vdash T : \diamond)$ and $T <: U$, then $(\Gamma \vdash U : \diamond)$.*

Proof. By induction on $<:$, using substitutivity (Lemma 8) to deal with the type parameters of reference types. □

[8] Recall that σ ranges over closing substitutions. See Appendix I for the definition of $\Gamma \vdash \sigma : \diamond$.

O.2 Properties of the Proof Theory

Lemma 11 (Well-Typedness). *If* $(\Gamma; v; \bar{F} \vdash G)$*, then* $(\Gamma \vdash v, \bar{F}, G : \diamond)$.

Proof. By inductions on the derivations. □

Lemma 12 (Weakening Validity of Boolean Expressions). *If* $\Gamma \models e$*,* $\Gamma \subseteq \Gamma'$ *and* $(\Gamma' \vdash \diamond)$*, then* $\Gamma' \models e$.

Proof. This holds because, by definition, $\Gamma \models e$ entails that e is true in all heaps that extend Γ_{hp}, and, moreover, the truth of e does not depend on variables outside $\mathsf{fv}(e)$. □

Lemma 13 (Weakening).

(a) *If* $(\Gamma; v; \bar{F} \vdash G)$*,* $\Gamma \subseteq \Gamma'$ *and* $(\Gamma' \vdash \diamond)$*, then* $(\Gamma'; v; \bar{F} \vdash G)$.
(b) *If* $(\Gamma; v; \bar{F} \vdash G)$ *and* $(\Gamma \vdash \bar{E} : \diamond)$*, then* $(\Gamma; v; \bar{F}, \bar{E} \vdash G)$.

Proof. By inductions on the derivation of $(\Gamma; v; \bar{F} \vdash G)$. □

Lemma 14 (Substitutivity for Validity of Boolean Expressions). *If* $(\Gamma[\bar{\pi}/\bar{x}] \vdash \bar{\pi} : \bar{T}[\bar{\pi}/\bar{x}])$ *and* $(\Gamma, \bar{x} : \bar{T} \models e)$*, then* $(\Gamma[\bar{\pi}/\bar{x}] \models e[\bar{\pi}/\bar{x}])$.

Proof. Let $(\Gamma[\bar{\pi}/\bar{x}] \vdash \bar{\pi} : T[\bar{\pi}/\bar{x}])$ and $(\Gamma, \bar{x} : \bar{T} \models e)$. Let $\Gamma' \supseteq_{\mathsf{hp}} \Gamma[\bar{\pi}/\bar{\alpha}]$, $(\Gamma'_{\mathsf{hp}} \vdash h : \diamond)$ and $(\Gamma' \vdash \sigma : \diamond)$. We need to show that $[\![e[\bar{\pi}/\bar{x}][\sigma]]\!]^h_\emptyset = \mathtt{true}$. To this end, let $\sigma' = \sigma[\bar{x} \mapsto \bar{\pi}[\sigma]]$. Note that $e[\bar{\pi}/\bar{x}][\sigma] = e[\sigma']$. Therefore, we are done if we can show that $[\![e[\sigma']]\!]^h_\emptyset = \mathtt{true}$. Let $\bar{y} = \mathsf{dom}(\Gamma \setminus \Gamma_{\mathsf{hp}})$. Let $\Gamma'' = (\Gamma'_{\mathsf{hp}}, \bar{y} : \Gamma(\bar{y}), \bar{x} : \bar{T})$. Because $(\Gamma, \bar{x} : \bar{T} \models e)$, we know that $(\Gamma, \bar{x} : \bar{T} \vdash e : \mathtt{bool})$, thus, $(\Gamma_{\mathsf{hp}} \vdash \diamond)$, thus, $\mathsf{fv}(\Gamma_{\mathsf{hp}}) = \emptyset$, thus, $\Gamma[\bar{\pi}/\bar{x}]_{\mathsf{hp}} = \Gamma_{\mathsf{hp}}$, thus, $\Gamma'' \supseteq_{\mathsf{hp}} (\Gamma, \bar{x} : \bar{T})$. Moreover, $(\Gamma''_{\mathsf{hp}} \vdash h : \diamond)$, because $\Gamma''_{\mathsf{hp}} = \Gamma'_{\mathsf{hp}}$ and $(\Gamma'_{\mathsf{hp}} \vdash h : \diamond)$. Because $(\Gamma, \bar{x} : \bar{T} \models e)$, it therefore suffices to show that $(\Gamma'' \vdash \sigma' : \diamond)$:

Let first $y \in \bar{y}$. Because $(\Gamma' \vdash \sigma : \diamond)$, we know that $(\Gamma'_{\mathsf{hp}} \vdash \sigma(y) : \Gamma'(y)[\sigma])$. We also know that $\Gamma'_{\mathsf{hp}} = \Gamma''_{\mathsf{hp}}$ and $\sigma(y) = \sigma'(y)$ and $\Gamma'(y)[\sigma] = \Gamma[\bar{\pi}/\bar{x}](y)[\sigma] = \Gamma(y)[\bar{\pi}/\bar{x}][\sigma] = \Gamma(y)[\sigma']$. Hence, $\Gamma''_{\mathsf{hp}} \vdash \sigma'(y) : \Gamma(y)[\sigma']$.

Let now $x \in \bar{x}$ such that $\sigma'(x) = \pi[\sigma]$ and $\Gamma''(x) = T$. We know that $(\Gamma' \vdash \sigma : \diamond)$ and $(\Gamma' \vdash \pi : T[\bar{\pi}/\bar{x}])$. By substitutivity (Lemma 8(c)), it follows that $(\Gamma'_{\mathsf{hp}} \vdash \pi[\sigma] : T[\bar{\pi}/\bar{x}][\sigma])$. We know that $\Gamma'_{\mathsf{hp}} = \Gamma''_{\mathsf{hp}}$ and $\pi[\sigma] = \sigma'(x)$ and $T[\bar{\pi}/\bar{x}][\sigma] = T[\sigma']$. Therefore, $\Gamma''_{\mathsf{hp}} \vdash \sigma'(x) : T[\sigma']$. □

Lemma 15 (Substitutivity). *If* $(\Gamma[\bar{\pi}/\bar{x}] \vdash \bar{\pi} : \bar{T}[\bar{\pi}/\bar{x}])$ *and* $(\Gamma, \bar{x} : \bar{T}; v; \bar{F} \vdash G)$*, then* $(\Gamma; v; \bar{F})[\bar{\pi}/\bar{x}] \vdash G[\bar{\pi}/\bar{x}]$.

Proof. By induction on the derivation of $(\Gamma, \bar{x} : \bar{T}; v; \bar{F} \vdash G)$. □

Lemma 16 (Specialization of Variable Types). *If* $(\Gamma, x : U; v; \bar{F} \vdash G)$*,* $(\Gamma, x : T \vdash \diamond)$ *and* $T <: U$*, then* $(\Gamma, x : T; v; \bar{F} \vdash G)$.

Proof. This follows from substitutivity (Lemma 15) and $(\Gamma, x : T \vdash x : U)$. □

Lemma 17 (Cut). *If* $(\Gamma; v; \bar{E} \vdash F)$ *and* $(\Gamma; v; F, \bar{G} \vdash H)$*, then* $(\Gamma; v; \bar{E}, \bar{G} \vdash H)$.

Proof. By induction on the derivation of $(\Gamma; v; F, \bar{G} \vdash H)$. □

Lemma 18 (`ispartof` is a Preorder).

(a) *If* $\Gamma \vdash v, F : \diamond$, *then* $(\Gamma; v; \texttt{true} \vdash F \texttt{ ispartof } F)$.
(b) $(\Gamma; v; \texttt{true} \vdash F \texttt{ ispartof } H)$ *is derivable from* $(\Gamma; v; \texttt{true} \vdash F \texttt{ ispartof } G)$ *and* $(\Gamma; v; \texttt{true} \vdash G \texttt{ ispartof } H)$.

The derivations only use the rules for $*$, $-\!*$ *and the identity rule.*

Proof. Straightforward natural deduction proofs. □

O.3 Properties of Hoare Triples

Lemma 19 (Well-Typedness).

(a) *If* $(\Gamma; v \vdash \{F\} hc \{G\})$, *then* $(\Gamma \vdash v, F, G : \diamond)$.
(b) *If* $(\Gamma; v \vdash \{F\} c : T \{G\})$, *then* $(\Gamma \vdash v, F, T, G : \diamond)$.

Proof. For *hc* by inspection of the last rule. For *c* by induction on the structure of *c*. □

Lemma 20 (Weakening).

(a) *If* $(\Gamma; v \vdash \{F\} hc \{G\})$, $\Gamma \subseteq \Gamma'$ *and* $(\Gamma' \vdash \diamond)$, *then* $(\Gamma'; v \vdash \{F\} hc \{G\})$.
(b) *If* $(\Gamma; v \vdash \{F\} c : T \{G\})$, $\Gamma \subseteq \Gamma'$ *and* $(\Gamma' \vdash \diamond)$, *then* $(\Gamma'; v \vdash \{F\} c : T \{G\})$.

Proof. For *hc* by inspection of the last rule. For *c* by induction on the structure of *c*. □

Lemma 21 (Substitutivity).

(a) *If* $(\Gamma[\bar{\pi}/\bar{x}] \vdash \bar{\pi} : \bar{T}[\bar{\pi}/\bar{x}])$ *and* $(\Gamma, \bar{x} : \bar{T}; v \vdash \{F\} hc \{G\})$, *then* $((\Gamma; v)[\bar{\pi}/\bar{x}] \vdash \{F[\bar{\pi}/\bar{x}]\} hc [\bar{\pi}/\bar{x}] \{G[\bar{\pi}/\bar{x}]\})$.
(b) *If* $(\Gamma[\bar{\pi}/\bar{x}] \vdash \bar{\pi} : \bar{T}[\bar{\pi}/\bar{x}])$ *and* $(\Gamma, \bar{x} : \bar{T}; v \vdash \{F\} c : U \{G\})$, *then* $((\Gamma; v)[\bar{\pi}/\bar{x}] \vdash \{F[\bar{\pi}/\bar{x}]\} c [\bar{\pi}/\bar{x}] : U[\bar{\pi}/\bar{x}] \{G[\bar{\pi}/\bar{x}]\})$.

Proof. For *hc* by inspection of the last rule. For *c* by induction on the structure of *c*. □

Lemma 22 (Logical Consequence). *If* $(\Gamma; v; F \vdash F')$ *and* $(\Gamma; v \vdash \{F'\} c : T \{G\})$, *then* $(\Gamma; v \vdash \{F\} c : T \{G\})$.

Proof. By induction on the structure of *c*. □

Lemma 23 (Subsumption). *If* $(\Gamma; v \vdash \{F\} c : T \{G\})$ *and* $T <: U$, *then* $(\Gamma; v \vdash \{F\} c : U \{G\})$.

Proof. By induction on the structure of *c*. □

Lemma 24 (Frame Lemma). *If* $(\Gamma; v \vdash \{F\} c : T \{(\texttt{ex } T' \alpha)(G)\})$, $\Gamma \vdash H : \diamond$ *and* $\mathsf{fv}(c) \cap \mathsf{fv}(H) \subseteq \mathsf{RdVar} \cup \mathsf{LogVar}$, *then* $(\Gamma; v \vdash \{F * H\} c : T \{(\texttt{ex } T' \alpha)(G * H)\})$.

Proof. By induction on the structure of *c*. □

Lemma 25 (Derived Rule for Bind). *If* $(\Gamma; o \vdash \{F\} c : T \{(\texttt{ex } T \alpha)(G)\})$, $T <: \Gamma(\ell)$ *and* $(\Gamma; p \vdash \{(\texttt{ex } T \alpha)(\alpha == \ell * G)\} c' : U \{H\})$, *then* $(\Gamma; o \vdash \{F\} \ell \leftarrow c; c' : U \{H\})$.

Proof. By induction on the structure of *c*. □

Lemma 26 (Derived Rule for Sequential Composition). *If* $(\Gamma; o \vdash \{F\} c : \texttt{void}\{G\})$ *and* $(\Gamma; o \vdash \{G\} c' : T \{H\})$, *then* $(\Gamma; o \vdash \{F\} c; c' : T \{H\})$.

Proof. This is a consequence of the derived rule for bind (Lemma 25). □

O.4 Properties of the Semantics

Lemma 27 (Expression Semantics Preserves Typings). *If* $(\Gamma \vdash e : T)$, $(\Gamma_{\mathsf{hp}} \vdash h : \diamond)$, $(\Gamma \vdash s : \diamond)$ *and* $[\![e]\!]^h_s = \mu$, *then* $(\Gamma \vdash \mu : T)$.

Proof. By induction on $(\Gamma \vdash e : T)$. □

Lemma 28 (Expression Have Values). *If* $(\Gamma \vdash e : T)$, $(\Gamma_{\mathsf{hp}} \vdash h : \diamond)$, $\mathsf{fv}(e) \subseteq \mathsf{dom}(h)$ *and* $(\Gamma \vdash s : \diamond)$, *then* $[\![e]\!]^h_s = \mu$ *for some some* μ.

Proof. By induction on $(\Gamma \vdash e : T)$. □

Lemma 29. *If* $[\![op]\!]^h(\bar{v}) = w$ *and* $h \leq h'$, *then* $[\![op]\!]^{h'}(\bar{v}) = w$.

Proof. Our conditions on $[\![op]\!]$ (see Appendix F) were upwards closure with respect to $\subseteq$ and invariance under field updates. If $h \leq h'$, then there exists h'' such that $h'' \subseteq h'$ and h'' can be obtained from h by a sequence of field updates. □

Lemma 30 (Resource Monotonicity).

(a) *If* $[\![e]\!]^h_s = \mu$ *and* $h \leq h'$, *then* $[\![e]\!]^{h'}_s = \mu$.
(b) *If* $(\Gamma \vdash \mathscr{E}; \mathscr{R}; s \models F)$, $\mathscr{R} \leq \mathscr{R}'$, $\Gamma \subseteq_{\mathsf{hp}} \Gamma'$ *and* $(\Gamma' \vdash \mathscr{R}', s, F : \diamond)$, *then* $(\Gamma' \vdash \mathscr{E}; \mathscr{R}'; s \models F)$.

Proof. Part (a) by induction on the structure of e, using Lemma 29. Part (b) by induction on the structure of F. The proof is very similar to the proof of Lemma 71 in [12]. □

Lemma 31 (Stack Invariance).

(a) *If* $s_{|\mathsf{fv}(e)} = s'_{|\mathsf{fv}(e)}$ *and* $[\![e]\!]^h_s = \mu$, *then* $[\![e]\!]^h_{s'} = \mu$.
(b) *If* $s_{|\mathsf{fv}(F)} = s'_{|\mathsf{fv}(F)}$, $\Gamma_{\mathsf{hp}} = \Gamma'_{\mathsf{hp}}$, $\Gamma_{|\mathsf{fv}(F)} = \Gamma'_{|\mathsf{fv}(F)}$, $(\Gamma' \vdash s' : \diamond)$ *and* $(\Gamma \vdash \mathscr{E}; \mathscr{R}; s \models F)$, *then* $(\Gamma' \vdash \mathscr{E}; \mathscr{R}; s' \models F)$.

Proof. Part (a) by induction on the structure of e. Part (b) by induction on the structure of F. The proof is very similar to the proof of Lemma 72 in [12]. □

Lemma 32 (Value Substitutivity for Semantics).

(a) $[\![e[v/\ell]]\!]^h_s = \mu$ *iff* $[\![e]\!]^h_{s[\ell \mapsto v]} = \mu$.
(b) *If* $(\Gamma \vdash v : T)$, $(\Gamma \vdash s : \diamond)$ *and* $(\Gamma, \ell : T \vdash F : \diamond)$, *then* $(\Gamma \vdash \mathscr{E}; \mathscr{R}; s \models F[v/\ell])$ *iff* $(\Gamma, \ell : T \vdash \mathscr{E}; \mathscr{R}; s[\ell \mapsto v] \models F)$.

Proof. Part (a) by induction on the structure of e. part (b) by induction on the structure of F. The proof is very similar to the proof of Lemma 73 in [12]. □

Lemma 33 (Expression Substitutivity for Semantics).

(a) *If* $[\![\pi_1]\!] = [\![\pi_2]\!]$, *then* $\pi[\pi_1/x] \simeq \pi[\pi_2/x]$ *and* $T[\pi_1/x] <: T[\pi_2/x]$.
(b) *If* $[\![e[e_1/x]]\!]^h_s = \mu$ *and* $[\![e_1]\!]^h_s = [\![e_2]\!]^h_s$, *then* $[\![e[e_2/x]]\!]^h_s = \mu$.
(c) *If* $(\Gamma \vdash \mathscr{E}; \mathscr{R}; s \models F[e_1/x])$, $h = \mathscr{R}_{\mathsf{hp}}$, $[\![e_1]\!]^h_s = [\![e_2]\!]^h_s$ *and* $(\Gamma \vdash F[e_2/x] : \diamond)$, *then* $(\Gamma \vdash \mathscr{E}; \mathscr{R}; s \models F[e_2/x])$.

Proof. Part (a) is a consequence of the injectivity of value semantics (Lemma 2). Parts (b) is shown by induction on the structure of e, part (c) by induction on the structure of F. The proof is very similar to the proof of Lemma 74 in [12]. □

Lemma 34 (Monotonicity of Initialized Sets).
If $(\Gamma \vdash \mathscr{E}; \mathscr{R}; s \models F)$, $\mathscr{R}' = (\mathscr{R}_{\mathsf{hp}}, \mathscr{R}_{\mathsf{perm}}, \mathscr{R}_{\mathsf{lock}}, \mathscr{R}_{\mathsf{fresh}}, \mathscr{R}_{\mathsf{init}} \cup \mathscr{I})$ *and* $\mathscr{I} \subseteq \mathsf{dom}(\Gamma)$, *then* $(\Gamma \vdash \mathscr{E}; \mathscr{R}'; s \models F)$.

Proof. By induction on the structure of F, making use of the syntactic restriction that the `initialized`-predicate must not occur in negative positions. □

P Preservation

Our preservation proof extends the preservation proof from [12] with proof cases for the synchronization primitives. We slightly modify our proof architecture from [12][9] to account for the structural rules, i.e., the rules (Con), (Exists), (Disj) and (Frame). To this end, we observe that we can normalize Hoare proofs for head commands as follows:

Lemma 35 (Proof Normalization). *If* $(\Gamma; v \vdash \{F\} hc \{G\})$ *is derivable, then it has a proof where every path to the proof goal ends in zero or more applications of (Con), (Exists) and (Disj), preceded by exactly one application of (Frame), preceded by a rule that is not a structural rule.*

Proof. We need to show that we can permute an application of (Frame) upwards, when preceded by (Con), (Exists) or (Disj). These permutations are straightforward to show. By associativity of $*$ we can condense a sequence of several (Frame) applications into a single (Frame) application. By neutrality of `true`, we can expand zero (Frame) applications into one (Frame) application. □

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

Proof.

(1) $ct : \diamond$ assumption
(2) $st : \diamond$ assumption
(3) $st \rightarrow st'$ assumption

An inspection of the reduction rules shows that st is of the following form, where o is the thread that the reduction rule "operates on" (i.e., for all rules but (Red Notify) the only thread on the reduction's left hand side whose components are explicitly named, and for (Red Notify) the thread whose head command is `notify()`):

(4) $st = \langle h, ts \mid o \,\mathsf{is}\, (s \,\mathsf{in}\, c) \rangle$

The proof of $(st : \diamond)$ ends in an application of (State), preceded by an application of (Cons Pool), preceded by an application of (Thread) for thread o. The rule (Thread) has the following premises:

[9] In [12], we used a variant of separation logic whose only structural rules were the logical consequence rule and the frame rule. We represented the auxiliary variable rule (Exists) syntactically by explicit existential unpacking, and omitted the rule of disjunction altogether.

(5) $\Gamma \vdash \sigma : \Gamma'$
(6) $\Gamma, \Gamma' \vdash s : \diamond$
(7) $\Gamma[\sigma] \vdash \mathscr{E}; \mathscr{R}; s \models F[\sigma]$
(8) $\Gamma, \Gamma'; r \vdash \{F\} c : \mathtt{void}\{G\}$

We split cases according to the shape of c. Unless c is of the form $c = hc; c'$, the reduction rule is one of (Red Dcl), (Red Fin Dcl) or (Red Return). For each of these cases the proof of $st' : \diamond$ is just like the corresponding case in the proof of Theorem 5 in [12]. We omit these proof cases here, and assume from this point on that c is of the form $c = hc; c'$:

(9) $c = hc; c'$
(10) $\Gamma, \Gamma'; r \vdash \{F\} hc \{F'\}$
(11) $\Gamma, \Gamma'; r \vdash \{F'\} c' : \mathtt{void}\{G\}$

Let $\mathscr{D}$ be the proof tree of $(\Gamma, \Gamma'; r \vdash \{F\} hc \{F'\})$. By Lemma 35, we may assume that each path to the root of $\mathscr{D}$ ends in a sequence of applications of (Con), (Exists) and (Disj), preceded by exactly one application of (Frame). We induct on the height of $\mathscr{D}$:

Case 1, *$\mathscr{D}$ ends in (Con):* In this case, we have:

(1.1) $\Gamma, \Gamma'; r; F \vdash H$
(1.2) $\Gamma, \Gamma'; r \vdash \{H\} hc \{H'\}$
(1.3) $\Gamma, \Gamma'; r; H' \vdash F'$

From $(\Gamma[\sigma] \vdash \mathscr{E}; \mathscr{R}; s \models F[\sigma])$ and $(\Gamma, \Gamma'; r; F \vdash H)$, it follows that:

(1.4) $\Gamma[\sigma] \vdash \mathscr{E}; \mathscr{R}; s \models H[\sigma]$

From $(\Gamma, \Gamma'; r; H' \vdash F')$ and $(\Gamma, \Gamma'; r \vdash \{F'\} c' : \mathtt{void}\{G\})$, it follows that:

(1.5) $\Gamma, \Gamma'; r \vdash \{H'\} c' : \mathtt{void}\{G\}$

The height of $(\Gamma, \Gamma'; r \vdash \{H\} hc \{H'\})$'s proof tree is one less than $\mathscr{D}$'s height. In the proof tree of $st : \diamond$, we replace (7) by (1.4), (10) by (1.2), and (11) by (1.5). The resulting tree is a proof tree of $st : \diamond$. By induction hypothesis we obtain $st' : \diamond$.

Case 2, *$\mathscr{D}$ ends in (Exists):* In this case, we have:

(2.1) $F = (\mathtt{ex}\ T\ \alpha)(H)$
(2.2) $\Gamma, \Gamma', \alpha : T; r \vdash \{H\} hc \{H'\}$
(2.3) $F' = (\mathtt{ex}\ T\ \alpha)(H')$

From $(\Gamma[\sigma] \vdash \mathscr{E}; \mathscr{R}; s \models F[\sigma])$, it follows that there is a π such that $(\Gamma[\sigma]_{\mathsf{hp}} \vdash \pi : T[\sigma])$ and $(\Gamma[\sigma] \vdash \mathscr{E}; \mathscr{R}; s \models H[\sigma][\pi/\alpha])$. Let $\Gamma'' = (\Gamma', \alpha : T)$ and $\sigma' = (\sigma, \pi/\alpha)$. Then:

(2.4) $\Gamma \vdash \sigma' : \Gamma''$
(2.5) $\Gamma, \Gamma'' \vdash s : \diamond$
(2.6) $\Gamma[\sigma'] \vdash \mathscr{E}; \mathscr{R}; s \models H[\sigma']$

From $(\Gamma, \Gamma''; r; H' \vdash F')$ and $(\Gamma, \Gamma'; r \vdash \{F'\} c' : \mathtt{void}\{G\})$, it follows that:

(2.7) $\Gamma, \Gamma''; r \vdash \{H'\} c' : \mathtt{void}\{G\}$

The height of $(\Gamma, \Gamma''; r \vdash \{H\} hc \{H'\})$'s proof tree is one less than $\mathscr{D}$'s height. In the proof tree of $st : \diamond$, we replace (5) by (2.4), (6) by (2.5), (7) by (2.6), (10) by (2.2), (11) by (2.7). The resulting tree is a proof tree of $st : \diamond$. By induction hypothesis, $st' : \diamond$.

Case 3, *$\mathscr{D}$ ends in (Disj):* In this case, we have:

(3.1) $F = H_1 \mid H_2$
(3.2) $\Gamma, \Gamma'; r \vdash \{H_1\} hc \{H_1'\}$
(3.3) $\Gamma, \Gamma'; r \vdash \{H_2\} hc \{H_2'\}$
(3.4) $F' = H_1' \mid H_2'$

From $(\Gamma[\sigma] \vdash \mathscr{E}; \mathscr{R}; s \models F[\sigma])$, it follows that $(\Gamma[\sigma] \vdash \mathscr{E}; \mathscr{R}; s \models H_1[\sigma])$ or $(\Gamma[\sigma] \vdash \mathscr{E}; \mathscr{R}; s \models H_2[\sigma])$. Without loss of generality, we assume the former:

(3.5) $\Gamma[\sigma] \vdash \mathscr{E}; \mathscr{R}; s \models H_1[\sigma]$

From $(\Gamma, \Gamma'; r; H_1' \vdash F')$ and $(\Gamma, \Gamma'; r \vdash \{F'\} c' : \texttt{void}\{G\})$, it follows that:

(3.6) $\Gamma, \Gamma'; r \vdash \{H_1'\} c' : \texttt{void}\{G\}$

The height of $(\Gamma, \Gamma''; r \vdash \{H_1\} hc \{H_1'\})$'s proof tree is at least one less than $\mathscr{D}$'s height. In the proof tree of $st : \diamond$, we replace (7) by (3.5), (10) by (3.2), (11) by (3.6). The resulting tree is a proof tree of $st : \diamond$. By induction hypothesis we obtain $st' : \diamond$.

Case 4, *$\mathscr{D}$ ends in (Frame) preceded by a non-structural rule:* We split this case into subcases according to the reduction rules. Most of these subcases are just like in [12], Theorem 5, and we omit them. The additional subcases are the ones for the synchronization primitives, and we show these in detail. We also show the case for object creation, because object creation has a stronger postcondition than in [12].

Let $\mathscr{R}^{ts}$ and $\mathscr{R}'$ be the resources that satisfy the thread pool ts and the resource invariants of the initialized, unlocked objects (premises of (Cons Pool) and (State)):

(4.1.1) $\mathscr{R}^{ts} \vdash ts : \diamond$
(4.1.2) $(\mathscr{R} * \mathscr{R}^{ts}) \# \mathscr{R}'$
(4.1.3) $\Gamma'' \vdash \mathscr{E}; \mathscr{R}'; \emptyset \models \circledast_{q \in \mathsf{ready}(\mathscr{R} * \mathscr{R}^{ts})} q.\texttt{inv}$

Case 4.1, *(Red Lock):*

$$\frac{(l(p) = \mathsf{free},\ l' = l[p \mapsto (1,o)]) \text{ or } (l(p) = (n,o),\ l' = l[p \mapsto (n+1,o)])}{\langle h, l, ts \mid o \text{ is } (s \text{ in } p.\texttt{lock}(); c') \rangle \rightarrow \langle h, l', ts \mid o \text{ is } (s \text{ in } c') \rangle}$$

In this case, we can further instantiate hc and st' as follows:

(4.1.1) $hc = p.\texttt{lock}()$
(4.1.2) $st' = \langle h, l', ts \mid o \text{ is } (s \text{ in } c') \rangle$
(4.1.3) $(l(p) = \mathsf{free},\ l' = l[p \mapsto (1,o)])$ or $(l(p) = (n,o),\ l' = l[p \mapsto (n+1,o)])$

We know that $\mathscr{D}$ ends in an application of (Frame), preceded by (Re-Lock) or (Lock).

Case 4.1.1, *(Re-Lock):* This case is straightforward. We omit the details.

Case 4.1.2, *(Lock):* In this case:

(4.1.2.1) $F = \texttt{Lockset}(\pi) * !(\pi \texttt{ contains } p) * p.\texttt{initialized} * H$
(4.1.2.2) $F' = \texttt{Lockset}(p \cdot \pi) * p.\texttt{inv} * H$

So we know that:

(4.1.2.3) $\Gamma[\sigma] \vdash \mathscr{E}; \mathscr{R}; s \models (\texttt{Lockset}(\pi) * !(\pi \texttt{ contains } p) * p.\texttt{initialized} * H)[\sigma]$

Because $(\Gamma[\sigma] \vdash \mathscr{E}; \mathscr{R}; s \models (\texttt{Lockset}(\pi) * !(\pi \texttt{ contains } p))[\sigma])$ contradicts $l(p) = (n,o)$, we know that:

(4.1.2.4) $l(p) = \mathsf{free},\ l' = l[p \mapsto (1,o)]$

We now split $\mathscr{R}$ into $\mathscr{R} = \mathscr{R}^1 * \mathscr{R}^2$ such that:

(4.1.2.5) $\Gamma[\sigma] \vdash \mathscr{E}; \mathscr{R}^1; s \models \texttt{Lockset}(\pi)[\sigma]$

(4.1.2.6) $\Gamma[\sigma] \vdash \mathscr{E}; \mathscr{R}^2; s \models H[\sigma]$

We need a resource that satisfies $\texttt{Lockset}(p \cdot \pi)[\sigma]$. To this end, we define:

(4.1.2.7) $\mathscr{R}^3 \triangleq (\emptyset, 0, \{o \mapsto \mathscr{R}^1_{\mathsf{lock}} \sqcup [p]\}, \emptyset, \mathscr{R}^1_{\mathsf{init}})$

By (4.1.2.5), we have $\mathscr{R}^1_{\mathsf{lock}}(o) = [\![\pi[\sigma]]\!]$. Thus, $\mathscr{R}^3_{\mathsf{lock}}(o) = [\![p \cdot \pi[\sigma]]\!]$. Thus:

(4.1.2.8) $\Gamma[\sigma] \vdash \mathscr{E}; \mathscr{R}^3; s \models \texttt{Lockset}(p \cdot \pi)[\sigma]$

Because $l(p) = \mathsf{free}$ and $\mathscr{R}'_{\mathsf{lock}} = \emptyset$, we know that $p \notin \bigsqcup\{\ (\mathscr{R}^2 * \mathscr{R}^{ts} * \mathscr{R}')_{\mathsf{lock}}(q) \mid q \in \mathsf{dom}((\mathscr{R}^2 * \mathscr{R}^{ts} * \mathscr{R}')_{\mathsf{lock}})\ \}$. Thus:

(4.1.2.9) $\mathscr{R}^3 \# (\mathscr{R}^2 * \mathscr{R}^{ts} * \mathscr{R}')$

Now, we need a resource that satisfies $p.\texttt{inv}[\sigma]$. We want to split this resource off $\mathscr{R}'$. Because $(\Gamma[\sigma] \vdash \mathscr{E}; \mathscr{R}; s \models p.\texttt{initialized})$, we know that $p \in (\mathscr{R} * \mathscr{R}^{ts})_{\mathsf{init}}$. We also know $l(p) = \mathsf{free}$. It follows that $p \in \mathsf{ready}(\mathscr{R} * \mathscr{R}^{ts})$. Therefore, we can split $\mathscr{R}'$ into $\mathscr{R}' = \mathscr{R}^p * \mathscr{R}''$ such that:

(4.1.2.10) $\Gamma'' \vdash \mathscr{E}; \mathscr{R}^p; \emptyset \models p.\texttt{inv}$

(4.1.2.11) $\Gamma'' \vdash \mathscr{E}; \mathscr{R}''; \emptyset \models \circledast_{q \in (\mathsf{ready}(\mathscr{R} * \mathscr{R}^{ts}) \setminus \{p\})} q.\texttt{inv}$

Now, the remaining problem is that the type environment Γ'' *in* (4.1.2.10) *differs from the type environments* $\Gamma[\sigma]$ *in* (4.1.2.6) *and* (4.1.2.8). Fortunately, we know (by premises of (Thread) and (State)) that $\Gamma_{\mathsf{hp}} \subseteq \mathsf{fst} \circ h = \Gamma''$, where h is the global heap. Therefore, we can union Γ and Γ''. Moreover, $\Gamma''[\sigma] = \Gamma''$, because types of object identifiers do not contain variables. So we can weaken (4.1.2.8), (4.1.2.10) and (4.1.2.6) to obtain:

(4.1.2.12) $(\Gamma \cup \Gamma'')[\sigma] \vdash \mathscr{E}; \mathscr{R}^3; s \models \texttt{Lockset}(p \cdot \pi)[\sigma]$

(4.1.2.13) $(\Gamma \cup \Gamma'')[\sigma] \vdash \mathscr{E}; \mathscr{R}^p; s \models p.\texttt{inv}$

(4.1.2.14) $(\Gamma \cup \Gamma'')[\sigma] \vdash \mathscr{E}; \mathscr{R}^2; s \models H[\sigma]$

Joining these three statements, we obtain:

(4.1.2.15) $(\Gamma \cup \Gamma'')[\sigma] \vdash \mathscr{E}; \mathscr{R}^3 * \mathscr{R}^p * \mathscr{R}^2; s \models F'[\sigma]$

In the proof tree of $st : \diamond$, we replace (7) by (4.1.2.15), and (8) by (11). The resulting tree is a proof tree of $st' : \diamond$. Note that we use the fact that $\mathsf{dom}(\mathscr{R}^p_{\mathsf{lock}}) = \emptyset$ (by premise of (State)) in order to establish the premise $\mathsf{dom}((\mathscr{R}^3 * \mathscr{R}^p * \mathscr{R}^2)_{\mathsf{lock}}) \subseteq \{o\}$ of (Thread).

Case 4.2, *(Red Unlock):*

$$\frac{l(p) = (n,o) \quad n = 1 \Rightarrow l' = l[p \mapsto \mathsf{free}] \quad n > 1 \Rightarrow l' = l[p \mapsto (n-1,o)]}{\langle h, l, ts \mid o\ \mathsf{is}\ (s\ \mathsf{in}\ p.\texttt{unlock}(); c')\rangle \rightarrow \langle h, l', ts \mid o\ \mathsf{is}\ (s\ \mathsf{in}\ c')\rangle}$$

In this case, we can further instantiate hc and st' as follows:

(4.2.1) $hc = p.\texttt{unlock}()$

(4.2.2) $st' = \langle h, l', ts \mid o\ \mathsf{is}\ (s\ \mathsf{in}\ c')\rangle$

(4.2.3) $l(p) = (n,o)$ and $(n = 1 \Rightarrow l' = l[p \mapsto \mathsf{free}])$ and $(n > 1 \Rightarrow l' = l[p \mapsto (n-1,o)])$

We know that $\mathscr{D}$ ends in an application of (Frame), preceded by (Re-Unlock) or (Unlock).

Case 4.2.1, *(Re-Unlock):* This case is straightforward. We omit the details.

Case 4.2.2, *(Unlock):* In this case:

(4.2.2.1) $F = \texttt{Lockset}(p \cdot \pi) * p.\texttt{inv} * H$
(4.2.2.2) $F' = \texttt{Lockset}(\pi) * H$

So we know that:

(4.2.2.3) $\Gamma[\sigma] \vdash \mathscr{E}; \mathscr{R}; s \models (\texttt{Lockset}(p \cdot \pi) * p.\texttt{inv} * H)[\sigma]$

We now split $\mathscr{R}$ into $\mathscr{R} = \mathscr{R}^1 * \mathscr{R}^2 * \mathscr{R}^p$ such that:

(4.2.2.4) $\Gamma[\sigma] \vdash \mathscr{E}; \mathscr{R}^1; s \models \texttt{Lockset}(p \cdot \pi)[\sigma]$
(4.2.2.5) $\Gamma[\sigma] \vdash \mathscr{E}; \mathscr{R}^2; s \models H[\sigma]$
(4.2.2.6) $\Gamma[\sigma] \vdash \mathscr{E}; \mathscr{R}^p; s \models p.\texttt{inv}$

From (4.2.2.4), we know that $\mathscr{R}^1_{\mathsf{lock}} = \mathscr{L} \cup \{o \mapsto [p] \sqcup [\![\pi]\!]\}$ for some $\mathscr{L}$. We define:

(4.2.2.7) $\mathscr{R}^3 \triangleq (\emptyset, 0, \mathscr{L}, \emptyset, \mathscr{R}_{\mathsf{init}})$

Clearly, $(\Gamma[\sigma] \vdash \mathscr{E}; \mathscr{R}^3; s \models \texttt{Lockset}(\pi)[\sigma])$. So we have:

(4.2.2.8) $\Gamma[\sigma] \vdash \mathscr{E}; \mathscr{R}^3 * \mathscr{R}^2; s \models F'[\sigma]$

We now show that $\mathscr{R}^p_{\mathsf{lock}} = \emptyset$. We know that $\mathsf{dom}(\mathscr{R}_{\mathsf{lock}}) \subseteq \{o\}$ (by premise of (Thread)) and $\mathsf{dom}(\mathscr{R}^1_{\mathsf{lock}}) \neq \emptyset$ (because $\mathscr{R}^1$ satisfies a `Lockset` predicate). From these facts it follows that $\mathscr{R}^1_{\mathsf{lock}} = \{o\}$ and $\mathscr{R}^p_{\mathsf{lock}} \subseteq \{o\}$. Because we also have $\mathscr{R}^1_{\mathsf{lock}} \# \mathscr{R}^p_{\mathsf{lock}}$, it follows that $\mathscr{R}^p_{\mathsf{lock}} = \emptyset$.

Knowing $\mathscr{R}^p_{\mathsf{lock}} = \emptyset$, we can derive:

$$\begin{array}{rcl} & \mathsf{ready}(\mathscr{R}^3 * \mathscr{R}^2 * \mathscr{R}^{ts}) & = \mathsf{ready}(\mathscr{R}^3 * \mathscr{R}^p * \mathscr{R}^2 * \mathscr{R}^{ts}) \\ \subseteq & \mathsf{ready}(\mathscr{R}^1 * \mathscr{R}^p * \mathscr{R}^2 * \mathscr{R}^{ts}) \cup \{p\} & = \mathsf{ready}(\mathscr{R} * \mathscr{R}^{ts}) \cup \{p\} \end{array}$$

In case $\mathsf{ready}(\mathscr{R}^3 * \mathscr{R}^2 * \mathscr{R}^{ts}) = \mathsf{ready}(\mathscr{R} * \mathscr{R}^{ts})$, the remainder of this proof is straightforward. So let us assume $p \in \mathsf{ready}(\mathscr{R}^3 * \mathscr{R}^2 * \mathscr{R}^{ts}) \setminus \mathsf{ready}(\mathscr{R} * \mathscr{R}^{ts})$. Then we have:

(4.2.2.9) $\Gamma'' \vdash \mathscr{E}; \mathscr{R}' * \mathscr{R}^p; \emptyset \models \circledast_{q \in \mathsf{ready}(\mathscr{R}^2 * \mathscr{R}^3 * \mathscr{R}^{ts})} q.\texttt{inv}$

In the proof tree of $st : \diamond$, we replace (7) by (4.2.2.8), (8) by (11), and (4.1.3) by (4.2.2.9). The resulting tree is a proof tree of $st' : \diamond$.

Case 4.3, *(Red No Op) for Commit:*

$$\frac{}{\langle h, l, ts \mid o \ \mathsf{is}\ (s \ \mathsf{in}\ p.\texttt{commit}; c')\rangle \rightarrow \langle h, l, ts \mid o \ \mathsf{is}\ (s \ \mathsf{in}\ c')\rangle}$$

In this case, we can further instantiate hc and st' as follows:

(4.3.1) $hc = p.\texttt{commit}$
(4.3.2) $st' = \langle h, ts \mid o \ \mathsf{is}\ (s \ \mathsf{in}\ c')\rangle$

We know that $\mathscr{D}$ ends in an application of (Frame), preceded by (Commit).

(4.3.3) $F = \texttt{Lockset}(\pi) * p.\texttt{inv} * p.\texttt{fresh} * H$
(4.3.4) $F' = \texttt{Lockset}(\pi) * !(\pi \ \texttt{contains}\ p) * p.\texttt{initialized} * H$

So we know that:

(4.3.5) $\Gamma[\sigma] \vdash \mathscr{E}; \mathscr{R}; s \models (\texttt{Lockset}(\pi) * p.\texttt{inv} * p.\texttt{fresh} * H)[\sigma]$

We now split $\mathscr{R}$ into $\mathscr{R} = \mathscr{R}^1 * \mathscr{R}^p * \mathscr{R}^2 * \mathscr{R}^3$ such that:

(4.3.6) $\Gamma[\sigma] \vdash \mathscr{E}; \mathscr{R}^1; s \models \texttt{Lockset(}\pi\texttt{)}[\sigma]$
(4.3.7) $\Gamma[\sigma] \vdash \mathscr{E}; \mathscr{R}^p; s \models p.\texttt{inv}$
(4.3.8) $\Gamma[\sigma] \vdash \mathscr{E}; \mathscr{R}^2; s \models p.\texttt{fresh}$
(4.3.9) $\Gamma[\sigma] \vdash \mathscr{E}; \mathscr{R}^3; s \models H[\sigma]$

Now we define new resources by adding p to the initialized sets:

(4.3.10) $\mathsf{init}_p(h, \mathscr{P}, \mathscr{L}, \mathscr{F}, \mathscr{I}) \triangleq (h, \mathscr{P}, \mathscr{L}, \mathscr{F}, \mathscr{I} \cup \{p\})$

By Lemma 34, we know that extending the initialized set preserves validity:

(4.3.11) $\mathsf{init}_p(\mathscr{R}^{ts}) \vdash ts : \diamond$
(4.3.12) $\Gamma'' \vdash \mathscr{E}; \mathsf{init}_p(\mathscr{R}'); \emptyset \models \circledast_{q \in \mathsf{ready}(\mathscr{R} * \mathscr{R}^{ts})} q.\texttt{inv}$
(4.3.13) $\Gamma[\sigma] \vdash \mathscr{E}; \mathsf{init}_p(\mathscr{R}^1); s \models \texttt{Lockset(}\pi\texttt{)}[\sigma]$
(4.3.14) $\Gamma[\sigma] \vdash \mathscr{E}; \mathsf{init}_p(\mathscr{R}^p); s \models p.\texttt{inv}$
(4.3.15) $\Gamma[\sigma] \vdash \mathscr{E}; \mathsf{init}_p(\mathscr{R}^3); s \models H[\sigma]$

By the resource axioms, we know that $\mathscr{R}_{\mathsf{fresh}} \cap \mathscr{R}_{\mathsf{init}} = \emptyset$. On the other hand, we have $p \in \mathscr{R}^2_{\mathsf{fresh}} \subseteq \mathscr{R}_{\mathsf{fresh}}$. Thus $p \notin \mathscr{R}_{\mathsf{init}}$, thus $p \notin \mathscr{R}^1_{\mathsf{init}}$. Another resource axiom says that $p \in \mathscr{R}^1_{\mathsf{lock}}$ implies $p \in \mathscr{R}^1_{\mathsf{init}}$. Thus $p \notin \mathscr{R}^1_{\mathsf{lock}}$, thus $p \notin \mathsf{init}_p(\mathscr{R}^1_{\mathsf{lock}})$, thus:

(4.3.16) $\Gamma[\sigma] \vdash \mathscr{E}; \mathsf{init}_p(\mathscr{R}^1 * \mathscr{R}^3); s \models (\texttt{Lockset(}\pi\texttt{)} * \texttt{!(}\pi\ \texttt{contains}\ p\texttt{)} * p.\texttt{initialized} * H)[\sigma]$

From (4.3.14) and (4.3.12), we obtain:

(4.3.17) $\Gamma'' \vdash \mathscr{E}; \mathsf{init}_p(\mathscr{R}') * \mathsf{init}_p(\mathscr{R}^p); \emptyset \models \circledast_{q \in \mathsf{ready}(\mathsf{init}_p(\mathscr{R}^1 * \mathscr{R}^3) * \mathsf{init}_p(\mathscr{R}^{ts}))} q.\texttt{inv}$

Furthermore, $\mathsf{init}_p(\mathscr{R}^p_{\mathsf{lock}}) = \emptyset$, because $\mathscr{R}^1_{\mathsf{lock}} = \{o\}$, $\mathscr{R}^p_{\mathsf{lock}} \subseteq \{o\}$, and $\mathscr{R}^1_{\mathsf{lock}} \# \mathscr{R}^p_{\mathsf{lock}}$.

In the proof tree of $st : \diamond$, we replace (7) by (4.3.16), (8) by (11), (4.1.1) by (4.3.11), and (4.1.3) by (4.3.17). The resulting tree is a proof tree of $st' : \diamond$.

Case 4.4, *(Red Wait):*

$$\frac{l(p) = (n, o) \quad l' = l[p \mapsto \mathsf{free}]}{\langle h, l, ts \mid o\ \mathsf{is}\ (s\ \mathsf{in}\ \ell\texttt{=}p.\texttt{wait()}; c')\rangle \to \langle h, l', ts \mid o\ \mathsf{is}\ (s\ \mathsf{in}\ p.\texttt{waiting(}n\texttt{)}; p.\texttt{resume(}n\texttt{)}; c')\rangle}$$

In this case, we can further instantiate hc and st' as follows:

(4.4.1) $c = \ell\texttt{=}p.\texttt{wait()}$
(4.4.2) $st' = \langle h, l', ts \mid o\ \mathsf{is}\ (s\ \mathsf{in}\ p.\texttt{waiting(}n\texttt{)}; c')\rangle$
(4.4.3) $l(p) = (n, o)$
(4.4.4) $l' = l[p \mapsto \mathsf{free}]$

We know that $\mathscr{D}$ ends in an application of (Frame), preceded by (Call). We look up `wait`'s specification in the `Thread` class, and obtain:

(4.4.5) $F = p\ \texttt{!=null} * \texttt{Lockset(}\pi\texttt{)} * \pi\ \texttt{contains}\ p * p.\texttt{inv} * H$
(4.4.6) $\ell \notin \mathsf{fv}(H)$
(4.4.7) $F' = \texttt{(ex void}\,\alpha\texttt{)(}\alpha\texttt{=}\ell * \texttt{Lockset(}\pi\texttt{)} * p.\texttt{inv)} * H$

So we know that:

(4.4.8) $\Gamma[\sigma] \vdash \mathscr{E}; \mathscr{R}; s \models (p\ \texttt{!=null} * \texttt{Lockset(}\pi\texttt{)} * \pi\ \texttt{contains}\ p * p.\texttt{inv} * H)[\sigma]$

Clearly, $(\Gamma, \Gamma'; r; F' \vdash \texttt{Lockset(}\pi\texttt{)} * p.\texttt{inv} * H)$. Therefore, we have:

(4.4.9) $\Gamma, \Gamma'; r \vdash \{\texttt{Lockset}(\pi) * p.\texttt{inv} * H\} c' : \texttt{void}\{G\}$

We now split $\mathscr{R}$ into $\mathscr{R} = \mathscr{R}^1 * \mathscr{R}^p * \mathscr{R}^2$ such that:

(4.4.10) $\Gamma[\sigma] \vdash \mathscr{E}; \mathscr{R}^1; s \models \texttt{Lockset}(\pi)[\sigma]$
(4.4.11) $\Gamma[\sigma] \vdash \mathscr{E}; \mathscr{R}^p; s \models p.\texttt{inv}$
(4.4.12) $\Gamma[\sigma] \vdash \mathscr{E}; \mathscr{R}^2; s \models H[\sigma]$

Because $l(p) = (n, o)$, we know that $\mathscr{R}^1_{\mathsf{lock}} = \mathscr{L} \cup \{o \mapsto \bar{p} \sqcup bag\}$, where $\bar{p}$ is the multiset with n occurrences of p, and bag does not contain p. Furthermore, because $\pi[\sigma]$ is closed, we know that it must be of the form $\pi[\sigma] \simeq p^n \cdot \pi'$ for some π' such that $[\![\pi']\!] = bag$. We define:

(4.4.13) $\mathscr{R}^3 \triangleq (\emptyset, 0, \mathscr{L} \cup \{o \mapsto bag\}, \emptyset, \mathscr{R}^1_{\mathsf{init}})$

Then we have:

(4.4.14) $\Gamma[\sigma] \vdash \mathscr{E}; \mathscr{R}^3; s \models \texttt{Lockset}(\pi')$

Because $p \in \mathscr{R}^1_{\mathsf{lock}}$ implies $p \in \mathscr{R}^1_{\mathsf{init}}$ (by one of the resource axioms), and because $\mathscr{R}^3_{\mathsf{init}} = \mathscr{R}^1_{\mathsf{init}}$, we have:

(4.4.15) $\Gamma[\sigma] \vdash \mathscr{E}; \mathscr{R}^3 * \mathscr{R}^2; s \models \texttt{Lockset}(\pi') * p.\texttt{initialized} * H[\sigma]$

By applying the rules (Resume), (Waiting) and (Frame) to (4.4.9), we obtain:

(4.4.16) $\Gamma, \Gamma'; r \vdash \begin{array}{c} \{\texttt{Lockset}(\pi[\sigma]) * p.\texttt{initialized} * H\} \\ p.\texttt{waiting}(n); p.\texttt{resume}(n); c' : \texttt{void} \\ \{G\} \end{array}$

From (4.1.3) and (4.4.11), we obtain:

(4.4.17) $\Gamma'' \vdash \mathscr{E}; \mathscr{R}' * \mathscr{R}^p; \emptyset \models \circledast_{q \in \mathsf{ready}(\mathscr{R}^2 * \mathscr{R}^3 * \mathscr{R}^{ts})} q.\texttt{inv}$

In the proof tree of $st : \diamond$, we replace (7) by (4.4.15), (8) by (4.4.16), and (4.1.3) by (4.4.17). The resulting tree is a proof tree of $st' : \diamond$.

Case 4.5, *(Red Notify):*

$$\frac{l(p) = (n, o)}{\langle h, l, ts \mid o \text{ is } (s \text{ in } \ell = p.\texttt{notify}(); c') \mid q \text{ is } (s_q \text{ in } p.\texttt{waiting}(n'); c_q) \rangle \rightarrow \langle h, l, ts \mid o \text{ is } (s \text{ in } c') \mid q \text{ is } (s_q \text{ in } c_q) \rangle}$$

This proof case is trivial, because in the specification of `notify` and the Hoare rule for `waiting`, the precondition implies the postcondition.

Case 4.6, *(Red Skip Notify):*

$$\frac{l(p) = (n, o)}{\langle h, l, ts \mid o \text{ is } (s \text{ in } \ell = p.\texttt{notify}(); c') \rangle \rightarrow \langle h, l, ts \mid o \text{ is } (s \text{ in } c') \rangle}$$

This proof case is trivial, because `notify`'s precondition implies its postcondition.

Case 4.7, *(Red Resume):*

$$\frac{l(p) = \mathsf{free} \quad l' = l[p \mapsto (n, o)]}{\langle h, l, ts \mid o \text{ is } (s \text{ in } p.\texttt{resume}(n); c') \rangle \rightarrow \langle h, l', ts \mid o \text{ is } (s \text{ in } c') \rangle}$$

This proof case is very similar to the proof case for (Red Lock).

Case 4.8, *(Red New):*

$$\frac{p \notin \mathsf{dom}(h) \quad h' = h[p \mapsto (C\texttt{<}\bar{\pi}\texttt{>}, \mathsf{init}(C\texttt{<}\bar{\pi}\texttt{>}))] \quad s' = s[\ell \mapsto p] \quad l' = l[o \mapsto \mathsf{free}]}{\langle h, l, ts \mid o \text{ is } (s \text{ in } \ell\texttt{=new } C\texttt{<}\bar{\pi}\texttt{>}; c')\rangle \rightarrow \langle h', l', ts \mid o \text{ is } (s' \text{ in } c')\rangle}$$

In this case, we can further instantiate hc and st' as follows:

(4.8.1) $hc = \ell\texttt{=new } C\texttt{<}\bar{\pi}\texttt{>}$
(4.8.2) $st' = \langle h', ts \mid o \text{ is } (s' \text{ in } c')\rangle$
(4.8.3) $h' = h[p \mapsto (C\texttt{<}\bar{\pi}\texttt{>}, \mathsf{init}(C\texttt{<}\bar{\pi}\texttt{>}))]$
(4.8.4) $s' = s[\ell \mapsto p]$

From the premises of (New), we obtain:

(4.8.5) $C\texttt{<}\bar{T}\ \bar{\alpha}\texttt{>} \in ct$
(4.8.6) $\Gamma, \Gamma' \vdash \bar{\pi} : \bar{T}[\bar{\pi}/\bar{\alpha}]$
(4.8.7) $C\texttt{<}\bar{\pi}\texttt{>} <: \Gamma(\ell)$
(4.8.8) $F' = F * \ell\texttt{.init} * C\ \texttt{classof}\ \ell * \circledast_{\Gamma(u)<:\texttt{Object}}\ \ell\ \texttt{!=}\ u * \ell\texttt{.fresh}$
(4.8.9) $\ell \notin F$

By substitutivity, $(\Gamma[\sigma] \vdash \bar{\pi}[\sigma] : \bar{T}[\bar{\pi}/\bar{\alpha}][\sigma])$. By premise of (Thread), $\mathsf{dom}(\Gamma') \cap \mathsf{cfv}(hc) = \emptyset$, i.e., $\mathsf{dom}(\sigma) \cap \mathsf{fv}(\bar{\pi}) = \emptyset$. Therefore, $(\Gamma[\sigma] \vdash \bar{\pi} : \bar{T}[\bar{\pi}/\bar{\alpha}])$. Thus:

(4.8.10) $\Gamma[\sigma] \vdash C\texttt{<}\bar{\pi}\texttt{>} : \diamond$

This observation is needed for well-typedness of the semantic entailments (4.8.11), (4.8.13) and (4.8.14) below. In $(\Gamma[\sigma] \vdash \mathscr{E}; \mathscr{R}; s \models F[\sigma])$, we can extend the environment (by Lemma 30) and the stack (by Lemma 31) to obtain:

(4.8.11) $\Gamma[\sigma], p : C\texttt{<}\bar{\pi}\texttt{>} \vdash \mathscr{E}; \mathscr{R}; s' \models F[\sigma]$

We now define:

(4.8.12) $\mathscr{R}^1 = (\{p \mapsto (C\texttt{<}\bar{\pi}\texttt{>}, \mathsf{init}(C\texttt{<}\bar{\pi}\texttt{>}))\}, 0, \emptyset, \{p\}, \mathscr{R}_{\mathsf{init}})$

Clearly, the following holds:

(4.8.13) $\Gamma[\sigma], p : C\texttt{<}\bar{\pi}\texttt{>} \vdash \mathscr{E}; \mathscr{R}^1; s' \models \ell\texttt{.init} * C\ \texttt{classof}\ \ell * \circledast_{\Gamma(u)<:\texttt{Object}}\ \ell\ \texttt{!=}\ u * \ell\texttt{.fresh}$

Hence, we have:

(4.8.14) $\Gamma[\sigma], p : C\texttt{<}\bar{\pi}\texttt{>} \vdash \mathscr{E}; \mathscr{R} * \mathscr{R}^1; s' \models F'[\sigma]$

In the proof tree of $st : \diamond$, we replace (7) by (4.8.14), and (8) by (11). The resulting tree is a proof tree of $st' : \diamond$. □

Q Data Race Freedom, Error Freedom and Partial Correctness

As corollaries of the preservation theorem, we obtain that programs do not "go wrong", and that they satisfy partial correctness. In this paper, programs go wrong if they reach a state that that has a data race, or where a thread dereferences `null`, of where a thread calls o`.wait()` or o`.notify()` without holding o's lock.

A *program* is a pair (ct, c) of a class table ct and a main program c. Let main be a distinguished object identifier that acts as a dummy receiver for the main program. *Verified programs* are defined by the following rule:

$$\frac{ct : \diamond \qquad \mathsf{main} : \texttt{Thread}; \mathsf{main} \vdash \{\texttt{true}\}c : \texttt{void}\{\texttt{true}\} \qquad \mathsf{main} \text{ does not occur in } c}{(ct, c) : \diamond}$$

We define the *initial state of main program c* as the state whose heap consist of the dummy main object, whose lock table is empty, and whose thread pool consists of a single thread with empty stack and command c:

$$\mathsf{init}(c) \triangleq \langle \{\mathsf{main} \mapsto (\texttt{Thread}, \emptyset)\}, \emptyset, \mathsf{main\ is}\ (\emptyset\ \mathsf{in}\ c)\rangle$$

Lemma 36. *If* $(ct, c) : \diamond$*, then* $\mathsf{init}(c) : \diamond$*.*

Proof. Straightforward check. □

Q.1 Data Race Freedom

A pair (hc, hc') of head commands is called a *data race* iff $hc = (o.f = v)$ and either $hc' = (o.f = v')$ or $hc' = (\ell = o.f)$ for some o, f, v, v', ℓ.

Theorem 3 (Verified Programs are Data Race Free). *If* $(ct, c) : \diamond$ *and* $\mathsf{init}(c) \rightarrow_{ct}^{*} \langle h, l, ts \mid o_1\ \mathsf{is}\ (s_1\ \mathsf{in}\ hc_1; c_1) \mid o_2\ \mathsf{is}\ (s_2\ \mathsf{in}\ hc_2; c_2)\rangle$*, then* (hc_1, hc_2) *is not a data race.*

Proof. Let $(ct, c) : \diamond$, $st = \langle h, l, ts \mid o_1\ \mathsf{is}\ (s_1\ \mathsf{in}\ hc_1; c_1) \mid o_2\ \mathsf{is}\ (s_2\ \mathsf{in}\ hc_2; c_2)\rangle$, and $\mathsf{init}(c) \rightarrow_{ct}^{*} st$. By $\mathsf{init}(c) : \diamond$ (Lemma 36) and preservation (Theorem 1), we know that $st : \diamond$. Suppose, towards a contradiction, that (hc_1, hc_2) is a data race. An inspection of the last rules of $(st : \diamond)$'s derivation reveals that there must then be resources $\mathscr{R}$, $\mathscr{R}'$ and a heap cell $o.f$ such that $\mathscr{R} \vdash o_1\ \mathsf{is}\ (s_1\ \mathsf{in}\ hc_1; c_1) : \diamond$, $\mathscr{R}' \vdash o_2\ \mathsf{is}\ (s_2\ \mathsf{in}\ hc_2; c_2) : \diamond$, $\mathscr{R} \# \mathscr{R}'$, $\mathscr{R}_{\mathsf{perm}}(o, f) = 1$ and $\mathscr{R}'_{\mathsf{perm}}(o, f) > 0$. But then $\mathscr{R}_{\mathsf{perm}}(o, f) + \mathscr{R}'_{\mathsf{perm}}(o, f) > 1$, in contradiction to $\mathscr{R} \# \mathscr{R}'$. □

Q.2 Null Error Freedom

A head command hc is called a *null error* iff $hc = (\ell = \texttt{null}.f)$ or $hc = (\texttt{null}.f = v)$ or $hc = (\ell = \texttt{null}.m(\bar{v}))$ or $hc = (\texttt{null.lock()})$ or $hc = (\texttt{null.unlock()})$ for some $\ell, f, v, m, \bar{\pi}, \bar{v}$.

Theorem 4 (Verified Programs are Null Error Free). *If* $(ct, c) : \diamond$ *and* $\mathsf{init}(c) \rightarrow_{ct}^{*} \langle h, l, ts \mid o\ \mathsf{is}\ (s\ \mathsf{in}\ hc; c)\rangle$*, then hc is not a null error.*

Proof. Let $(ct, c) : \diamond$, $st = \langle h, ts \mid o\ \mathsf{is}\ (s\ \mathsf{in}\ hc; c)\rangle$, and $\mathsf{init}(c) \rightarrow_{ct}^{*} st$. By $\mathsf{init}(c) : \diamond$ (Lemma 36) and preservation (Theorem 1), we know that $st : \diamond$. Suppose, towards a contradiction, that hc is a null error. Then $hc = (\ell = \texttt{null}.f)$ or $hc = (\texttt{null}.f = v)$ or $hc = (\ell = \texttt{null}.m(\bar{\pi})\bar{v})$ or $hc = (\texttt{null.lock()})$ of $hc = (\texttt{null.unlock()})$.

Suppose first that $hc = (\ell = \texttt{null}.f)$. An inspection of the last rules of $(st : \diamond)$'s derivation reveals that there must then be $\Gamma, \mathscr{E}, \mathscr{R}, s, \pi, u$ such that either $\Gamma \vdash \mathscr{E}; \mathscr{R}; s \models \texttt{PointsTo(null}.f, \pi, u\texttt{)}$ holds. But this is false, by definition of $\models$.

Suppose now that $hc = (\texttt{null}.f = v)$ An inspection of the last rules of $(st : \diamond)$'s derivation reveals that there must then be Γ, $\mathscr{E}$, $\mathscr{R}$, s, T such that $\Gamma \vdash \mathscr{E}; \mathscr{R}; s \models \texttt{PointsTo(null}.f, 1, T\texttt{)}$. But this is false, by definition of $\models$.

Suppose now that $hc = (\ell = \texttt{null}.m(\bar{v}))$. An inspection of the last rules of $(st : \diamond)$'s derivation reveals that there must then be Γ, $\mathscr{E}$, $\mathscr{R}$, s such that $\Gamma \vdash \mathscr{E}; \mathscr{R}; s \models \texttt{null != null}$, which is obviously false.

Suppose now that $hc = (\texttt{null.lock()})$. An inspection of the last rules of $(st : \diamond)$'s derivation reveals that there must then be $\Gamma, \mathscr{E}, \mathscr{R}, s, e$ such that either *(1)* $\Gamma \vdash \mathscr{E}; \mathscr{R}; s \models$

`null.initialized` (in the case where the last rule is (Lock)) or *(2)* $\Gamma \vdash \mathscr{E};\mathscr{R};s \models$ `Lockset(null` $\cdot\, e$`)` (in the case where the last rule is (Re-Lock)). However, both cases are impossible. The first case is impossible because $\Gamma \vdash \mathscr{E};\mathscr{R};s \models$ `null.initialized` means that `null` $\in \mathscr{R}_{\mathsf{init}}$. But this is impossible, because $\mathscr{R}_{\mathsf{init}} \subseteq \mathsf{ObjId}$, and `null` is not an object id. As for the second case, $\Gamma \vdash \mathscr{E};\mathscr{R};s \models$ `Lockset(null` $\cdot\, e$`)` means that `null` $\in \mathscr{R}_{\mathsf{lock}}(o)$. But this is impossible because $\mathscr{R}_{\mathsf{lock}}(o) \in \mathsf{Bag}(\mathsf{ObjId})$, and `null` is not an object id.

Suppose now that $hc =$ (`null.unlock()`). An inspection of $(st : \diamond)$'s derivation reveals that there must be Γ, $\mathscr{E}$, $\mathscr{R}$, s, e such that $\Gamma \vdash \mathscr{E};\mathscr{R};s \models$ `Lockset(null` $\cdot$ `null` $\cdot\, e$`)` (in the case where the last rule is (Re-Unlock)) or $\Gamma \vdash \mathscr{E};\mathscr{R};s \models$ `Lockset(null` $\cdot\, e$`)` (in the case where the last rule is (Unlock)). Both cases are impossible, because locksets are bags of object ids, and `null` is not an object id. □

Q.3 No Illegal Monitor States

Java throws an `IllegalMonitorStateException` at runtime whenever a thread calls o`.wait()` or o`.notify()` without holding o's lock. Our verification system prevents such runtime errors statically.

An *illegal monitor state* is of the form $\langle h, l, ts \mid p \text{ is } (s \text{ in } hc; c)\rangle$ where *(1)* $l(o) = \mathsf{free}$ or $l(o) = (q,n), q \neq p$, and *(2)* $hc = \ell$`=`o`.wait()` or $hc = \ell$`=`o`.notify()`.

Theorem 5 (No Illegal Monitor States). *If* $(ct,c) : \diamond$ *and* $\mathsf{init}(c) \rightarrow_{ct}^{*} st$*, then* st *is not an illegal monitor state.*

Proof. This follows because the preconditions of `wait` and `notify` require that the receiver is locked. □

Q.4 Partial Correctness

Theorem 6 (Partial Correctness). *Suppose* $\mathscr{E} = \mathscr{F}_{ct}(\mathscr{E})$*. If* $(ct,c) : \diamond$ *and* $\mathsf{init}(c) \rightarrow_{ct}^{*} \langle h, l, ts \mid o \text{ is } (s \text{ in } \texttt{assert(}F\texttt{)}; c)\rangle$*, then* $(\Gamma \vdash \mathscr{E};\mathscr{R};s \models F[\sigma])$ *such that* $\mathscr{R}_{\mathsf{hp}} = h$ *for some* Γ*, and* $\sigma \in \mathsf{LogVar} \rightharpoonup \mathsf{SpecVal}$*.*

Proof. Let $(ct,c) : \diamond$, $st = \langle h, l, ts \mid o \text{ is } (s \text{ in } \texttt{assert(}F\texttt{)}; c)\rangle$, and $\mathsf{init}(c) \rightarrow_{ct}^{*} st$. By $\mathsf{init}(c) : \diamond$ (Lemma 36) and preservation (Theorem 1), we know that $st : \diamond$. An inspection of the last rules of $(st : \diamond)$'s derivation reveals that there must then be Γ, $\sigma \in \mathsf{LogVar} \rightharpoonup \mathsf{SpecVal}$ such that $(\Gamma \vdash \mathscr{E};\mathscr{R};s \models F[\sigma])$ and $\mathscr{R}_{\mathsf{hp}} = h$. □