

PERMISSION-BASED SEPARATION LOGIC FOR MULTITHREADED JAVA PROGRAMS*

AFSHIN AMIGHI^a, CHRISTIAN HAACK^b, MARIEKE HUISMAN^c, AND CLÉMENT HURLIN^d

^{a,c} University of Twente, The Netherlands
e-mail address: a.amighi@utwente.nl, marieke.huisman@ewi.utwente.nl

^b aicas GmbH, Karlsruhe, Germany
e-mail address: christian.haack@aicas.de

^d Prove & Run, Paris, France
e-mail address: clement.hurlin@provenrun.com

ABSTRACT. This paper presents a program logic for reasoning about multithreaded Java-like programs with dynamic thread creation, thread joining and reentrant object monitors. The logic is based on concurrent separation logic. It is the first detailed adaptation of concurrent separation logic to a multithreaded Java-like language.

The program logic associates a unique static access permission with each heap location, ensuring exclusive write accesses and ruling out data races. Concurrent reads are supported through fractional permissions. Permissions can be transferred between threads upon thread starting, thread joining, initial monitor entrances and final monitor exits. In order to distinguish between initial monitor entrances and monitor reentrances, auxiliary variables keep track of multisets of currently held monitors. Data abstraction and behavioral subtyping are facilitated through abstract predicates, which are also used to represent monitor invariants, preconditions for thread starting and postconditions for thread joining. Value-parametrized types allow to conveniently capture common strong global invariants, like static object ownership relations.

The program logic is presented for a model language with Java-like classes and interfaces, the soundness of the program logic is proven, and a number of illustrative examples are presented.

2012 ACM CCS: [Theory of computation]: Semantics and reasoning—Program reasoning—Program verification.

Key words and phrases: Program Verification, Java, Multithreaded Programs, Separation Logic.

* This work was funded in part by the 6th Framework programme of the EC under the MOBIUS project IST-FET-2005-015905 (Haack, Hurlin and Huisman) and ERC grant 258405 for the VerCors project (Huisman and Amighi).

^{a,c} Amighi and Huisman are supported by ERC grant 258405 for the VerCors project.

^b Part of the work done while the author was at Radboud University Nijmegen, Netherlands.

^c Part of the work done while the author was at INRIA Sophia Antipolis – Méditerranée, France.

^d Part of the work done while the author was at INRIA Sophia Antipolis – Méditerranée, France, and visiting the University of Twente, Netherlands; and then at INRIA – Bordeaux Sud-Ouest, France, Microsoft R&D, France and IRISA/Université de Rennes 1, France.

1. INTRODUCTION

1.1. Motivation and Context. In the last decade, researchers have spent great efforts on developing advanced program analysis tools for popular object-oriented programming languages, like Java or C#. Such tools include software model-checkers [VHB⁺03], static analysis tools for data race and deadlock detection [NAW06, NPSG09], type-and-effect systems for atomicity [FQ03, AFF06], and program verification tools based on interactive theorem proving [Hui01].

A particularly successful line of research is concerned with static contract checking tools, based on Hoare logic. Examples include ESC/Java [FLL⁺02] — a highly automatic, but deliberately unsound, tool based on a weakest precondition calculus and a SMT solver, the Key tool [BHS07] — a sound verification tool for Java programs based on dynamic logic and symbolic execution, and Spec# [BDF⁺04] — a sound modular verification tool for C# programs that achieves modular soundness by imposing a dynamic object ownership discipline. While still primarily used in academics, these tools are mature and usable enough, so that programmers other than the tool developers can employ them for constructing realistic, verified programs. A restriction, however, is that support for concurrency in static contract checking tools is still limited. Because most real-world applications written in Java or C# are multithreaded, this limitation is a serious obstacle for bringing assertion-based verification to the real world. Support for concurrency is therefore the most important next step for this technique.

What makes verification of shared-variable concurrent programs difficult is the possibility of thread interference. Any assertion that has been established by one thread can potentially be invalidated by any other thread at any time. Some traditional program logics for shared-variable concurrency, *e.g.*, Owicki-Gries [OG75] or Jones’s rely-guarantee method [Jon83], account for thread interference in the most general way. Unfortunately, the generality of these logics makes them tedious to use, perhaps even unsuitable as a practical foundation for verifying Java-like programs. In comparison to these logics, Hoare’s logics for conditional critical regions [Hoa72] and monitors [Hoa74] are much simpler, because they rely on syntactically enforceable synchronization disciplines that limit thread interference to a few synchronization points (see [And91] for a survey).

Because Java’s main thread synchronization mechanism is based on monitors, Hoare’s logic for monitors is a good basis for the verification of Java-like programs. Unfortunately, however, a safe monitor synchronization discipline cannot be enforced syntactically for Java. This is so, because Java threads typically share heap memory including possibly aliased variables. Recently, O’Hearn [O’H07] has generalized Hoare’s logic to programming languages with heap. To this end, he extended *separation logic* [IO01, Rey02], a new program logic, which had previously been used for reasoning about sequential pointer programs. *Concurrent separation logic (CSL)* [O’H07, Bro04] enforces correct synchronization of heap accesses *logically*, rather than *syntactically*. Logical enforcement of correct synchronization has the desirable consequence that all CSL-verified programs are guaranteed to be data-race free.

CSL has since been extended in various directions to make it more suitable to reason about more complex concurrent programs. For instance, Bornat and others have combined separation logic with permission accounting in order to support concurrent reads [BOCP05], while Gotsman et al. [GBC⁺07] and Hobor et al. [HAN08] have generalized concurrent separation logic to cope with Posix-style threads and locks, that is they can reason about dynamic allocation of locks and threads.

In this paper, we further adapt CSL and its extensions, to make it suitable to reason about a Java-like language. This requires several challenges to be addressed:

- Firstly, in Java locks are reentrant, dynamically allocated, and stored on the heap, and thus can be aliased. Reasoning about *storable locks* has been addressed before by Gotsman et al. [GBC⁺07] and Hobor et al. [HAN08], however these approaches do not generalise to reentrant locks. Supporting reentrant locks has important advantages, as they can avoid deadlocks due to attempted reentrancy. Such deadlocks would, for instance, occur when synchronized methods call synchronized methods on the current self: a very common call-pattern in Java. Therefore, any practical reasoning method for concurrent Java programs needs to provide support to reason about lock reentrancy.
- Secondly, Java threads are based on thread identifiers (represented by thread objects) that are dynamically allocated on the heap, can be stored on the heap and can be aliased. Additionally, a join-operation that is parametrized by a thread identifier allows threads to wait for the termination of other threads. A crucial difference with Posix threads is that Java threads can be joined multiple times, and the logic has to cater for this possibility.
- Finally, Java has a notifying mechanism to wake threads up while waiting for a lock. This is an efficient mechanism to allow threads to exchange information about the current shared state, without the need for continuous polling. A reasoning technique for Java thus should support this wait-notify mechanism.

The resulting proof system supports Java’s main concurrency primitives: dynamically created threads and monitors that can be stored on the heap, thread joining (possibly multiple times), monitor reentrancy, and the wait-notify mechanism. Furthermore, the proof system is carefully integrated into a Java-like type system, enriched with value-parametrized types. The resulting formal system allows reasoning about multithreaded programs written in Java. Since the use of Java is widespread (*e.g.*, internet applications, mobile phones and smart cards), this is an important step towards reasoning about realistic multi-threaded software.

1.2. Separation Logic Informally. Before discussing our contribution in detail, we first informally present the features of separation logic that are most important for this paper.

1.2.1. *Formulas as Access Tickets.* Separation logic [Rey02] combines the usual logical operators with the points-to predicate $x.f \mapsto v$ and the resource conjunction $F * G$.

The predicate $x.f \mapsto v$ has a *dual purpose*: firstly, it asserts that the object field $x.f$ contains data value v and, secondly, it represents a *ticket* that grants permission to access the field $x.f$. This is formalized by separation logic’s Hoare rules for reading and writing fields (where $x.f \mapsto _$ is short for $(\exists v)(x.f \mapsto v)$):

$$\{x.f \mapsto _ \} x.f = v \{x.f \mapsto v \} \quad \{x.f \mapsto v \} y = x.f \{x.f \mapsto v * v == y \}$$

The crucial difference to standard Hoare logic is that both these rules have a precondition of the form $x.f \mapsto _$: this formula functions as an *access ticket* for $x.f$. It is important that tickets are not duplicable: one ticket is not the same as two tickets! Intuitively, the formula $F * G$ represents two access tickets F and G to *separate* parts of the heap. In other words, the part of the heap that F permits to access is *disjoint* from the part of the heap that G permits to access. As a consequence, separation logic’s $*$ implicitly excludes interfering heap accesses through aliases: this is why the Hoare rules shown above are sound. It is noteworthy that given two objects \mathbf{a} and \mathbf{b} with field \mathbf{x} , the assertion $\mathbf{a}.\mathbf{x} \mapsto _ * \mathbf{b}.\mathbf{x} \mapsto _$

does not mean the same as $\mathbf{a.x} \mapsto _ \wedge \mathbf{b.x} \mapsto _$: the first assertion implies that \mathbf{a} and \mathbf{b} are distinct, while the second assertion can be satisfied even if \mathbf{a} and \mathbf{b} are aliases.

1.2.2. *Local Reasoning.* A crucial feature of separation logic is that it allows local reasoning, as expressed by the (Frame) rule:

$$\frac{\{F\}c\{F'\}}{\{F * G\}c\{F' * G\}} \text{ (Frame)}$$

This rule expresses that given a command c that only accesses the part of the heap described by F , one can reason locally about command c ((Frame)’s premise) and deduce something globally, i.e., in the context of a bigger heap $F * G$ ((Frame)’s conclusion). In this rule, G is called the frame and represents the part of the heap unaffected by executing c . It is important that the (Frame) rule can be added to our verification rules without harming soundness, because it enables modular verification, and in particular it allows one to verify method calls. When calling a method, from its specification one can identify the (small) part of the heap accessed by that method and use the frame rule to establish that the rest of the heap is left unaffected.

1.3. **Contributions.** Using the aspects of separation logic described above, we have developed a sound (but not complete) program logic for a concurrent language with Java’s main concurrency primitives. Our logic combines separation logic for Java [Par05] with fraction-based permissions [Boy03]. This results in an expressive and flexible logic, which can be used to verify many realistic applications. The logic ensures the absence of data races, but is not overly restrictive, as it allows concurrent reads. This subsection summarizes our system and highlights our contributions; for a detailed comparison with existing approaches, we refer to Section 5.

Because of the use of fraction-based permissions, as proposed by Boyland [Boy03], our program logic prevents data races, but allows multiple threads to read a location simultaneously. Permissions are fractions in the interval $(0, 1]$. Each access to the heap is associated with a permission. If a thread has full permission (*i.e.*, with value 1) to access a location, it can write this location, because the thread is guaranteed to have exclusive access to it. If a thread has a partial permission (less than 1), it can read a location. However, since other threads might also have permission to read the same location, a partial permission does not allow to write a location. Soundness of the approach is ensured by the guarantee that the total permissions to access a location are never more than 1.

Permissions can be transferred from one thread to another upon thread creation and thread termination. If a new thread is forked, the parent thread transfers the necessary permissions to this new thread (and thus the creating thread abandons these permissions, to avoid permission duplication). Once a thread terminates, its permissions can be transferred to the remaining threads. The mechanism for doing this in Java is by joining a thread: if a thread t joins another thread u , it blocks until u has terminated. After this, t can take hold of u ’s permissions. In order to soundly account for permissions upon thread joining, a special join-permission is used. Only threads that hold (a fraction of) this join-permission can take hold of (the same fraction of) the permissions that have been released by the terminating thread. Note that, contrary to Posix threads, Java threads allow multiple joiners of the same thread, and our logic supports this. For example, the logic can verify programs where

multiple threads join the same thread t in order to gain shared read-access to the part of the heap that was previously owned by t .

Just as in O’Hearn’s approach [O’H07], locks are associated with so-called resource invariants. If a thread acquires a lock, it may assume the lock’s resource invariant and obtain access to the resource invariant’s footprint (i.e., to the part of the heap that the resource invariant depends upon).

If a thread releases a lock, it has to establish the lock’s resource invariant and transfers access to the resource invariant’s footprint back to the lock. Previous variants of concurrent separation logic prohibit threads to acquire locks that they already hold. In contrast, Java’s locks are reentrant, and our program logic supports this. To this end, the logic distinguishes between initial lock entries and lock reentries. Permissions are transferred upon initial lock entries only, but not upon reentries.

Unfortunately, distinguishing between initial lock entries and reentries is not well-supported by separation logic. The problem is that this distinction requires proving that, upon initial entry, a lock does not alias any currently held locks. Separation logic, however, is designed to avoid depending on such global aliasing constraints, and consequently does not provide good support for reasoning about such. Fortunately, our logic includes a rich type system that can be used towards proving global aliasing constraints in many cases. The type system features value-parametrized types, which naturally extend Java’s type system that already includes generic types. Value parameters are used for static type checking and static verification only, thus, do not change the dynamic semantics of Java. Value-parametrized types can be useful in many ways. For instance, in [HH09] we use them to distinguish read-only iterators from read-write iterators. Value-parametrized types can also express static object ownership relations, as done in parametric ownership type systems (e.g., [CPN98, CD02]). Similar ownership type systems have been used in program verification systems to control aliasing (e.g, [Mül02]). In Section 4.6, we use type-based ownership towards proving the correctness of a fine-grained lock-coupling algorithms with our verification rules for reentrant locks. The type-based ownership relation serves to distinguish initial lock entries from lock reentries.

To allow the inheritance of resource invariants, we use abstract predicates as introduced in Parkinson’s object-oriented separation logic [Par05]. Abstract predicates hide implementation details from clients but allow class implementers to use them. Abstract predicates are highly appropriate to represent resource invariants: in class `Object` a resource invariant with empty footprint is defined, and each subclass can extend this resource invariant to depend on additional fields.

1.4. Earlier Papers and Overview. This paper is based on several earlier papers, presenting parts of the proof system. The logic to reason about dynamic threads was presented at AMAST 2008 [HH08a], the logic to reason about reentrant locks was presented at APLAS 2008 [HHH08]. However, compared to these earlier papers, the system has been unified and streamlined. In addition, novel specifications and implementations of sequential and parallel merge sort illustrate the approach. The work as it is presented here is adapted from a part of Hurlin’s PhD thesis [Hur09].

The remainder of this paper is organized as follows. Section 2 presents the Java-like language, permission-based separation logic and basic proof rules for single-threaded programs. Section 3 extends this to multithreaded programs with dynamic thread creation and termination, while Section 4 adds reentrant locks. Finally, Sections 5 and 6 discuss related

work, future work and conclusions. The complete soundness proof for the system can be found in Hurlin’s PhD thesis [Hur09].

2. THE SEQUENTIAL JAVA-LIKE LANGUAGE

This section presents a *sequential* Java-like (programming and specification) language that models core features of Java: mutable fields, inheritance and method overriding, and interfaces. Notice that we strongly base our work here on Parkinson’s thesis [Par05] and in particular reuse his notion of abstract predicate. Later sections will extend the language with Java-like concurrency primitives. Sequential programs written in the Java-like language can be specified and verified with separation logic. However, to simplify the presentation of the program logic, we assume that Java expressions are written in a form so that all intermediate results are assigned to local read-only variables (cf. e.g., [CWM99, SWM00, JW06, PB08]).

2.1. Syntax. The language distinguishes between read-only variables $\iota \in \text{RdVar}$, read-write variables $\ell \in \text{RdWrVar}$, and logical variables $\alpha \in \text{LogVar}$. Method parameters (including **this**) are always read-only, and local variables can be both read-only or read-write. Logical variables can only occur in specifications and types. We treat read-only variables specially, because their use often avoids the need for syntactical side conditions in the proof rules (see Section 2.4.2). The model language also includes class identifiers (**ClassId**), interface identifiers (**IntId**), field identifiers (**FieldId**), method identifiers (**MethId**) and predicate identifiers (**PredId**). Object identifiers (**ObjId**) are used in the operational semantics, but must not occur in source programs. Variable **Var** is the union of **RdVar**, **RdWrVar** and **LogVar**. In addition, type identifiers are defined as the union of **ClassId** and **IntId**.

Figure 1 defines syntax of our Java-like language. (**Open**) values are integers, booleans, object identifiers, **null**, and read-only variables. *Open values* are values that are not variables. Initially, *specifications values* range over logical variables and values; this will be extended in subsequent sections. To simplify later developments, our grammar for writing programs imposes that (1) every intermediate result is assigned to a local variable and (2) the right hand sides of assignments contain no read-write variables. Since interfaces and classes can be parametrized with specification values, object types are of the form $t\langle\bar{\pi}\rangle$. We introduce two special operators: **instanceof** and **classof**, where C **classof** v tests whether C is v ’s dynamic class. Note that these last two operators depend on object types, as stored on the heap. Classes do not have constructors: fields are initialized to a default value when objects are created. Later, for clarity, methods that act as constructors are called **init**. *Abstract predicates* [Par05, PB05] and class *axioms* are part of our specification language. Interfaces may declare abstract predicates and classes may implement them by providing concrete definitions as separation logic formulas. Appendix A defines syntactic functions to lookup fields, axioms, method types and bodies, and predicate types and bodies.

To write method contracts, we use *intuitionistic* separation logic [IO01, Rey02, Par05]. Contrary to classical separation logic, intuitionistic separation logic admits weakening i.e., it is invariant under heap extension. Informally, this means that one can ”forget” a part of the state, which makes it appropriate for garbage-collected languages.

The *resource conjunction* $F * G$ (a.k.a *separating conjunction*) expresses that resources F and G are independently available: using either of these resources leaves the other one

| | | |
|---|---|---|
| $n \in \text{Int}$ | $u, v, w \in \text{OpenVal} ::= \text{null} \mid n \mid b \mid o \mid \iota$ | $b \in \text{Bool} = \{\text{true}, \text{false}\}$ |
| Val | $= \text{OpenVal} \setminus \text{RdVar}$ | $\pi \in \text{SpecVal} ::= \alpha \mid v$ |
| $T, U, V, W \in \text{Type}$ | $::= \text{void} \mid \text{int} \mid \text{bool} \mid \text{perm} \mid t \langle \bar{\pi} \rangle$ | |
| $op \supseteq \{=, !, \&, \}$ | $\cup \{C \text{ classof} \mid C \in \text{ClassId}\} \cup \{\text{instanceof } T \mid T \in \text{Type}\}$ | |
| $e \in \text{Exp}$ | $::= \pi \mid \ell \mid op(\bar{e})$ | |
| $fd ::= T f;$ | field declarations | |
| $pd ::= \text{pred } P \langle \bar{T} \bar{\alpha} \rangle = F;$ | predicate definitions | |
| $ax ::= \text{axiom } F;$ | class axioms | |
| $md ::= \langle \bar{T} \bar{\alpha} \rangle \text{spec } U \ m(\bar{V} \bar{\iota}) \{c\}$ | methods (scope of $\bar{\alpha}, \bar{\iota}$ is $\bar{T}, \text{spec}, U, \bar{V}, c$) | |
| $\text{spec} ::= \text{requires } F; \text{ensures } F;$ | pre/postconditions | |
| $F \in \text{Formula}$ | specification formulas | |
| $cl \in \text{Class} ::= \text{class } C \langle \bar{T} \bar{\alpha} \rangle \text{ext } U \text{impl } \bar{V} \{fd^* pd^* ax^* md^*\}$ | classes (scope of $\bar{\alpha}$ is $\bar{T}, U, \bar{V}, fd^*, pd^*, ax^*, md^*$) | |
| $pt ::= \text{pred } P \langle \bar{T} \bar{\alpha} \rangle;$ | predicate types | |
| $mt ::= \langle \bar{T} \bar{\alpha} \rangle \text{spec } U \ m(\bar{V} \bar{\iota})$ | method types (scope of $\bar{\alpha}, \bar{\iota}$ is $\bar{T}, \text{spec}, U, \bar{V}$) | |
| $int \in \text{Interface} ::= \text{interface } I \langle \bar{T} \bar{\alpha} \rangle \text{ext } \bar{U} \{pt^* ax^* mt^*\}$ | interfaces (scope of $\bar{\alpha}$ is $\bar{T}, \bar{U}, pt^*, ax^*, mt^*$) | |
| $c \in \text{Cmd} ::= v \mid T \ell; c \mid T \iota = \ell; c \mid hc; c$ | | |
| $hc \in \text{HeadCmd} ::= \ell = v \mid \ell = op(\bar{v}) \mid \ell = v.f \mid v.f = v \mid \ell = \text{new } C \langle \bar{\pi} \rangle \mid \ell = v.m(\bar{v}) \mid \text{if } (v) \{c\} \text{else} \{c\} \mid \text{while } (e) \{c\}$ | | |
| $lop \in \{*, \text{-*}, \&, \}$ | $qt \in \{\text{ex}, \text{fa}\}$ | $\kappa \in \text{Pred} ::= P \mid P @ C$ |
| $F \in \text{Formula} ::= e \mid \text{PointsTo}(e.f, \pi, e) \mid \pi.\kappa \langle \bar{\pi} \rangle \mid F \text{lop } F \mid (qt \ T \ x)(F)$ | | |

FIGURE 1. Sequential Java-Like Language (JLL)

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* $F \text{-*} G$ (a.k.a. *linear implication* or *magic wand*) means "consume F yielding G ". Resource $F \text{-*} G$ permits to trade resource F to receive resource G in return. Most related work omit the magic wand. Parkinson and Bierman [PB05] entirely prohibit predicate occurrences in negative positions (i.e., to the left of an odd number of implications). We allow negative dependencies of predicate P on predicate Q as long as Q does not depend on P (cyclic predicate dependencies must be positive). We include it, because it can be added without any difficulties, and we found it useful to specify some typical programming patterns. Blom and Huisman show how the magic wand is used to state loop invariants over pointer data structures [BH13], while Haack and Hurlin use the magic wand to capture the behaviour of the iterator [HH09]. To avoid a proof theory with bunched contexts (see Section 2.4.1), we omit the \Rightarrow -implication between heap formulas (and did not need it in later examples).

The *points-to predicate* $\text{PointsTo}(e.f, \pi, v)$ is a textual representation for $e.f \xrightarrow{\pi} v$ [BOCP05]. Superscript π must be a fractional permission [Boy03] i.e., a fraction $\frac{1}{2^n}$ ($n \geq 0$) in the interval $(0, 1]$. The *predicate application* $\pi.\kappa \langle \bar{\pi} \rangle$ 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* $\pi.P@C<\bar{\pi}>$ represents the $*$ -conjunction of P 's predicate extensions in C 's superclasses, up to and including C . The *unqualified predicate* $\pi.P<\bar{\pi}>$ is equivalent to $\pi.P@C<\bar{\pi}>$, where C is π 's dynamic class. We allow predicates with missing parameters: semantically, missing parameters are existentially quantified. Predicate definitions can be preceded by an optional `public` modifier. The role of the `public` modifier is to export the definition of a predicate *in a given class* to clients (see e.g., the predicates in class `List` in the merge sort example in Section 2.6). For additional usage and formal definitions of `public`, we refer to [Hur09, §3.2.1] and Sections 3.5 and 4.6.

To be able to make mutable and immutable instances of the same class, it is crucial to allow parametrization of objects and predicates by permissions. For this, we include a special type `perm` for fractional permissions. Because class parameters are instantiated by specification values, we extend specification values with fractional permissions. Fractional permissions are represented symbolically: 1 represents itself, and if symbolic fraction π represents concrete fraction fr then `split(π)` represents $\frac{1}{2} \cdot fr$.

$$\pi \in \text{SpecVal} \quad ::= \quad \dots \mid 1 \mid \text{split}(\pi) \mid \dots$$

Quantified formulas have the shape $(qt \ T \ \alpha) (F)$, where qt is a universal or existential quantifier, α is a variable whose scope is formula F , and T is α 's type. Because specification values π and expressions e may contain logical variables α , quantified variables can appear in many positions: as type parameters; as the first, third, and fourth parameter in `PointsTo` predicates¹; as predicate parameters etc.

Class and interface declarations define *class tables* ($ct \subseteq \text{Interface} \cup \text{Class}$) ordered by *subtyping*. We write $\text{dom}(ct)$ for the set of all type identifiers declared in ct . *Subtyping* is defined in a standard way.

We define several convenient *derived forms* for specification formulas:

$$\begin{aligned} \text{PointsTo}(e.f, \pi, T) &\triangleq (\text{ex } T \ \alpha) (\text{PointsTo}(e.f, \pi, \alpha)) \\ \text{Perm}(e.f, \pi) &\triangleq (\text{ex } T \ \alpha) (\text{PointsTo}(e.f, \pi, \alpha)) \quad \text{where } T \text{ is } e.f\text{'s type} \\ F \text{ ** } G &\triangleq (F \text{ -* } G) \ \& \ (G \text{ -* } F) \\ F \text{ assures } G &\triangleq F \text{ -* } (F \text{ * } G) \\ F \text{ ispartof } G &\triangleq G \text{ -* } (F \text{ * } (F \text{ -* } G)) \end{aligned}$$

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

2.2. Operational Semantics. The operational semantics of our language is fairly standard, except that the state does not contain a call stack, but only a single store to keep track of the current receiver. It operates on states, consisting of a heap (`Heap`), a command (`Cmd`), and a stack (`Stack`). Section 3 will extend the state, to cope with multithreaded programs. Given a heap h and an object identifier o , we write $h(o)_1$ to denote o 's dynamic

¹Note that we forbid to quantify over the second parameter of `PointsTo` predicates, i.e., the field name. This is intentional, because this would complicate `PointsTo`'s semantics. We found this not to be a restriction, because we did not need this kind of quantification in any of our examples.

type and $h(o)_2$ to denote o 's store. We use the following abbreviation for field updates: $h[o.f \mapsto v] = h[o \mapsto (h(o)_1, h(o)_2[f \mapsto v])]$. For initial states, we define function init to denote a newly initialized object. Initially, the heap and the stack are empty.

Heap, Stack and State:

$$\begin{array}{l} \text{ObjStore} = \text{FieldId} \rightarrow \text{Val} \quad h \in \text{Heap} = \text{ObjId} \rightarrow \text{Type} \times \text{ObjStore} \\ s \in \text{Stack} = \text{RdWrVar} \rightarrow \text{Val} \quad st \in \text{State} = \text{Heap} \times \text{Cmd} \times \text{Stack} \end{array}$$

The semantics of values, operators and specification values are standard. The formal semantics of the built-in operators is presented in A.2 and the formal semantics of specification values is defined in A.3. In addition, we allow one to use any further built-in operator that satisfies the following two axioms:

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

The first of these axioms ensures that operators are invariant under heap extensions. The second axiom ensures that operators do not depend on values stored on the heap.

Auxiliary syntax for method call and return. We introduce a derived form, $\ell \leftarrow c; c'$ that assigns the result of a computation c to variable ℓ . In its definition, we write $\text{fv}(c)$ for the set of free variables of c . Furthermore, we make use of some auxiliary syntax $\ell = \mathbf{return}(v); c$. This construct is not meant to be used in source programs. Its purpose is to mark method-return-points in intermediate program states. The extra \mathbf{return} syntax allows us to associate a special proof rule with the post-state of method calls that characterizes this state. Technically, these definitions are chosen to support Lemma 2.3, which is central for dealing with call/return in the preservation proof.

$$\begin{array}{l} \ell \leftarrow v; c \triangleq \ell = \mathbf{return}(v); c \\ \ell \leftarrow (T \ell'; c); c' \triangleq T \ell'; \ell \leftarrow c; c' \quad \text{if } \ell' \notin \text{fv}(c') \text{ and } \ell' \neq \ell \\ \ell \leftarrow (T \iota = \ell'; c); c' \triangleq T \iota = \ell'; \ell \leftarrow c; c' \quad \text{if } \iota \notin \text{fv}(c') \\ \ell \leftarrow (hc; c); c' \triangleq hc; \ell \leftarrow c; c' \\ c ::= \dots \mid \ell = \mathbf{return}(v); c \mid \dots \end{array}$$

Restriction: This clause must not occur in source programs.

We can now also define sequential composition of commands as follows:

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

Small-step reduction. The state reduction relation \rightarrow_{ct} is given with respect to a class table ct . Where it is clear from the context, we omit the subscript ct . The complete set of the rules are defined in A.4, here we only discuss the most important cases.

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

$$\begin{array}{l} (\text{Red Dcl}) \quad \ell \notin \text{dom}(s) \quad s' = s[\ell \mapsto \text{df}(T)] \\ \quad \langle h, T \ell; c, s \rangle \rightarrow \langle h, c, s' \rangle \\ (\text{Red Fin Dcl}) \quad s(\ell) = v \quad c' = c[v/\iota] \\ \quad \langle h, T \iota = \ell; c, s \rangle \rightarrow \langle h, c', s \rangle \\ (\text{Red New}) \quad o \notin \text{dom}(h) \quad h' = h[o \mapsto (C\langle \bar{\pi} \rangle, \text{initStore}(C\langle \bar{\pi} \rangle))] \quad s' = s[\ell \mapsto o] \\ \quad \langle h, \ell = \mathbf{new} C\langle \bar{\pi} \rangle; c, s \rangle \rightarrow \langle h', c, s' \rangle \end{array}$$

(Red Call) $h(o)_1 = C\langle\bar{\pi}\rangle$ $\text{mbody}(m, C\langle\bar{\pi}\rangle) = (\iota_0; \bar{v}).c_m$ $c' = c_m[o/\iota_0, \bar{v}/\bar{v}]$
 $\langle h, \ell = o.m(\bar{v}); c, s \rangle \rightarrow \langle h, \ell \leftarrow c'; c, s \rangle$

In (Red Dcl), read-write variables are initialized to a default value. In (Red Fin Dcl), declaration of read-only variables is handled by substituting the right-hand side's value for the newly declared variable in the continuation. In (Red New), the heap is extended to contain a new object. In (Red Call), ι_0 is the formal method receiver and \bar{v} are the formal method parameters. Like for declaration of read-only variables, both the formal method receiver and the formal method parameters are substituted by the actual receiver and the actual method parameters.

2.3. Validity of Resource Formulas.

2.3.1. *Augmented heaps.* To define validity of our resource formulas, we augment the heap with a permission table. *Augmented heaps* \mathcal{H} as models of our formulas, range over the set AugHeap with a binary relation $\# \subseteq \text{AugHeap} \times \text{AugHeap}$ (the *compatibility relation*) and a partial binary operator $* : \# \rightarrow \text{AugHeap}$ (the *augmented heap joining operator*) that is associative and commutative. Concretely, augmented heaps are pairs $\mathcal{H} = (h, \mathcal{P})$ of a *heap* h and a *permission table* $\mathcal{P} \in \text{ObjId} \times \text{FieldId} \rightarrow [0, 1]$. To prove soundness of the verification rules for field updates and allocating new objects, augmented heaps must satisfy the following axioms:

- (a) $\mathcal{P}(o, f) > 0$ for all $o \in \text{dom}(h)$ and $f \in \text{dom}(h(o)_2)$.
- (b) $\mathcal{P}(o, f) = 0$ for all $o \notin \text{dom}(h)$ and all f .

Axiom (a) ensures that the (partial) heap h only contains cells that are associated with strictly positive permissions. Axiom (b) ensures that all unallocated objects have minimal permissions (with respect to the augmented heap order presented below).

Each of the two augmented heap components defines $\#$ (compatibility) and $*$ (joining) operators. *Heaps* are compatible if they agree on shared object types and memory content:

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

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

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

Permission tables join by point-wise addition: $(\mathcal{P} * \mathcal{P}')(o) \triangleq \mathcal{P}(o) + \mathcal{P}'(o)$, where compatibility ensures that the sums never exceed 1, i.e., $\mathcal{P} \# \mathcal{P}'$ iff $(\forall o)(\mathcal{P}(o) + \mathcal{P}'(o) \leq 1)$.

We define projection operators: $(h, \mathcal{P})_{\text{hp}} \triangleq h$ and $(h, \mathcal{P})_{\text{perm}} \triangleq \mathcal{P}$. Moreover, ordering on heaps, permission tables, and augmented heaps are defined as follows:

$$\begin{aligned} h \leq h' &\triangleq (\exists h'')(h * h'' = h') && : h \text{ contains less memory cells than } h' \\ \mathcal{P} \leq \mathcal{P}' &\triangleq (\exists \mathcal{P}'')(\mathcal{P} * \mathcal{P}'' = \mathcal{P}') && : \mathcal{P}' \text{'s permissions are less than } \mathcal{P}' \text{'s permissions} \\ \mathcal{H} \leq \mathcal{H}' &\triangleq (\exists \mathcal{H}'')(\mathcal{H} * \mathcal{H}'' = \mathcal{H}') && : \mathcal{H}' \text{'s components are all less than } \mathcal{H}' \text{'s components} \end{aligned}$$

$$\begin{array}{l}
 \Gamma \vdash \mathcal{E}; (h, \mathcal{P}); s \models e \quad \text{iff } \llbracket e \rrbracket_s^h = \mathbf{true} \\
 \Gamma \vdash \mathcal{E}; (h, \mathcal{P}); s \models \mathbf{PointsTo}(e, f, \pi, e') \quad \text{iff } \begin{cases} \llbracket e \rrbracket_s^h = o, h(o)_2(f) = \llbracket e' \rrbracket_s^h, \\ \text{and } \llbracket \pi \rrbracket \leq \mathcal{P}(o, f) \end{cases} \\
 \Gamma \vdash \mathcal{E}; \mathcal{H}; s \models \mathbf{null}.\kappa\langle \bar{\pi} \rangle \quad \text{iff } \mathbf{true} \\
 \Gamma \vdash \mathcal{E}; \mathcal{H}; s \models o.P@C\langle \bar{\pi} \rangle \quad \text{iff } \begin{cases} \mathcal{H}_{\text{hp}}(o)_1 <: C\langle \bar{\pi}' \rangle \text{ and} \\ \mathcal{E}(P@C)(\bar{\pi}', \mathcal{H}, o, \bar{\pi}) = 1 \end{cases} \\
 \Gamma \vdash \mathcal{E}; \mathcal{H}; s \models o.P\langle \bar{\pi} \rangle \quad \text{iff } \begin{cases} (\exists \bar{\pi}'')(\mathcal{H}_{\text{hp}}(o)_1 = C\langle \bar{\pi}' \rangle \text{ and} \\ \mathcal{E}(P@C)(\bar{\pi}', \mathcal{H}, o, (\bar{\pi}, \bar{\pi}'')) = 1 \end{cases} \\
 \Gamma \vdash \mathcal{E}; \mathcal{H}; s \models F * G \quad \text{iff } \begin{cases} (\exists \mathcal{H}_1, \mathcal{H}_2)(\mathcal{H} = \mathcal{H}_1 * \mathcal{H}_2, \\ \Gamma \vdash \mathcal{E}; \mathcal{H}_1; s \models F \text{ and} \\ \Gamma \vdash \mathcal{E}; \mathcal{H}_2; s \models G) \end{cases} \\
 \Gamma \vdash \mathcal{E}; \mathcal{H}; s \models F -* G \quad \text{iff } \begin{cases} (\forall \Gamma' \supseteq_{\text{hp}} \Gamma, \mathcal{H}') (\\ \mathcal{H} \# \mathcal{H}' \text{ and } \Gamma' \vdash \mathcal{E}; \mathcal{H}'; s \models F \\ \Rightarrow \Gamma' \vdash \mathcal{E}; \mathcal{H} * \mathcal{H}'; s \models G) \end{cases} \\
 \Gamma \vdash \mathcal{E}; \mathcal{H}; s \models F \& G \quad \text{iff } \Gamma \vdash \mathcal{E}; \mathcal{H}; s \models F \text{ and } \Gamma \vdash \mathcal{E}; \mathcal{H}; s \models G \\
 \Gamma \vdash \mathcal{E}; \mathcal{H}; s \models F \mid G \quad \text{iff } \Gamma \vdash \mathcal{E}; \mathcal{H}; s \models F \text{ or } \Gamma \vdash \mathcal{E}; \mathcal{H}; s \models G \\
 \Gamma \vdash \mathcal{E}; \mathcal{H}; s \models (\mathbf{ex } T \alpha)(F) \quad \text{iff } \begin{cases} (\exists \pi)(\Gamma_{\text{hp}} \vdash \pi : T \text{ and} \\ \Gamma \vdash \mathcal{E}; \mathcal{H}; s \models F[\pi/\alpha]) \end{cases} \\
 \Gamma \vdash \mathcal{E}; \mathcal{H}; s \models (\mathbf{fa } T \alpha)(F) \quad \text{iff } \begin{cases} (\forall \Gamma' \supseteq_{\text{hp}} \Gamma, \mathcal{H}' \geq \mathcal{H}, \pi)(\\ \Gamma'_{\text{hp}} \vdash \mathcal{H}'_{\text{hp}} : \diamond \text{ and } \Gamma'_{\text{hp}} \vdash \pi : T \\ \Rightarrow \Gamma' \vdash \mathcal{E}; \mathcal{H}'; s \models F[\pi/\alpha]) \end{cases}
 \end{array}$$

FIGURE 2. Meaning of formulas

2.3.2. *Meaning of Formulas.* To define the meaning of predicates, the notion of predicate environments is used. A predicate environment \mathcal{E} maps predicate identifiers to concrete heap predicates. Following Parkinson [Par05], it is defined as a least fixed point of an endofunction \mathcal{F}_{ct} on predicate environments. We do not detail its definition further, but instead refer to Parkinson's thesis.

An augmented heap \mathcal{H} is well-formed under typing environment Γ , *i.e.*, $(\Gamma \vdash \mathcal{H} : \diamond)$, whenever the heap and the permission table are well-formed, *i.e.* $\Gamma \vdash \mathcal{H}_{\text{hp}} : \diamond$ and $\mathcal{P}(o, f) > 0$ implies $o \in \text{dom}(\Gamma)$. Furthermore, given formula F and stack s , we say $(\Gamma \vdash \mathcal{E}, \mathcal{H}, s, F : \diamond)$ whenever the predicate environment is a least fixed point, and the augmented heap, stack, and formula are well-formed, *i.e.*, $\Gamma \vdash \mathcal{H} : \diamond$, $\Gamma \vdash s : \diamond$, and $\Gamma \vdash F : \diamond$, respectively². Now we define a forcing relation of the form $\Gamma \vdash \mathcal{E}; \mathcal{H}; s \models F$, which intuitively expresses that if $\Gamma \vdash \mathcal{E}; \mathcal{H}; s \models F$ holds, then the augmented heap \mathcal{H} is a state that is described by F . The relation $(\Gamma \vdash \mathcal{E}; \mathcal{H}; s \models F)$ is the unique subset of $(\Gamma \vdash \mathcal{E}, \mathcal{H}, s, F : \diamond)$ that satisfies the clauses in Figure 2.

2.4. **Verification.** This section first presents the proof theory, and next, Hoare triples to verify Java-like programs are introduced.

²All typing judgments are defined in A.5.

2.4.1. *Proof Theory.* As usual, Hoare triples use a logical consequence judgment. We define logical consequence proof-theoretically. The proof theory has two judgments:

$$\begin{array}{ll} \Gamma; v; \bar{F} \vdash G & G \text{ is a logical consequence of the } * \text{-conjunction of } \bar{F} \\ \Gamma; v \vdash F & F \text{ is an axiom} \end{array}$$

where \bar{F} is a *multiset* of formulas, and 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 (in an implementation, we would use an external and dedicated theorem prover to decide this relation) but instead we define as validity over all *closing substitutions*. Let σ range over *closing substitutions*, i.e., elements of $\text{Var} \rightarrow \text{Val}$:

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

$$\text{ClosingSubst}(\Gamma) \triangleq \{ \sigma \mid \Gamma \vdash \sigma : \diamond \}$$

We say that a heap h is *total* iff for all o in $\text{dom}(h)$ and all $f \in \text{dom}(\text{fld}(h(o)_1))$, $f \in \text{dom}(h(o)_2)$. Then we have: $\text{Heap}(\Gamma) \triangleq \{ h \mid \Gamma_{\text{hp}} \vdash h : \diamond \text{ and } h \text{ is total} \}$. Now, we define $\Gamma \models e$ as follows:

$$\Gamma \models e \quad \text{iff} \quad \left\{ \begin{array}{l} \Gamma \vdash e : \text{bool} \text{ and} \\ \forall \Gamma' \supseteq_{\text{hp}} \Gamma, h \in \text{Heap}(\Gamma'), \sigma \in \text{ClosingSubst}(\Gamma') : (\llbracket e[\sigma] \rrbracket_{\emptyset}^h = \text{true}) \end{array} \right.$$

Natural Deduction Rules. The logical consequence judgment of our Hoare logic is defined in a standard way based on the natural deduction calculus of (*affine*) *linear logic* [Wad93], which coincides with BI’s natural deduction calculus [OP99] on our restricted set of logical operators. The complete list is presented in A.6.

Axioms. In addition to the logical consequence judgment, sound *axioms* capture additional properties of our model. These additions do not harm soundness, as shown by Theorem 2.1 below. Table 1 presents the different axioms that we use:

- (Split/Merge) regulates permission accounting (where v denotes the current receiver and $\frac{\pi}{2}$ abbreviates `split`(π)).
- (Open/Close) allows predicate receivers to toggle between predicate names and predicate definitions (where $-$ as defined in A.1 – `pbody`($o.P\langle\bar{\pi}'\rangle, C\langle\bar{\pi}\rangle$) looks up $o.P\langle\bar{\pi}'\rangle$ ’s definition in the type $C\langle\bar{\pi}\rangle$ and returns its body F together with $C\langle\bar{\pi}\rangle$ ’s direct superclass $D\langle\bar{\pi}''\rangle$): Note that the current receiver, as represented on the left of the \vdash , has to match the predicate receiver on the right. This rule is the only reason why our logical consequence judgment tracks the current receiver. Note that $P@C$ may have more parameters than $P@D$: following Parkinson [Par05] we allow subclasses to extend predicate arities.
- (Missing Parameters) expresses that missing predicate parameters are existentially quantified.

| | |
|--|----------------------|
| $\Gamma; v \vdash \text{PointsTo}(e.f, \pi, e') \text{ ** } \left(\begin{array}{c} \text{PointsTo}(e.f, \frac{\pi}{2}, e') \\ * \\ \text{PointsTo}(e.f, \frac{\pi}{2}, e') \end{array} \right)$ | (Split/Merge) |
| $(\Gamma \vdash v : C \langle \bar{\pi}'' \rangle \wedge \text{pbody}(v.P \langle \bar{\pi}, \bar{\pi}' \rangle, C \langle \bar{\pi}'' \rangle) = F \text{ ext } D \langle \bar{\pi}''' \rangle) \Rightarrow \Gamma; v \vdash v.P @ C \langle \bar{\pi}, \bar{\pi}' \rangle \text{ ** } (F * v.P @ D \langle \bar{\pi} \rangle)$ | (Open/Close) |
| $\Gamma; v \vdash \pi.P \langle \bar{\pi} \rangle \text{ ** } (\text{ex } \bar{T} \bar{\alpha}) (\pi.P \langle \bar{\pi}, \bar{\alpha} \rangle)$ | (Missing Parameters) |
| $\Gamma; v \vdash \pi.P @ C \langle \bar{\pi} \rangle \text{ ispartof } \pi.P \langle \bar{\pi} \rangle$ | (Dynamic Type) |
| $C \preceq D \Rightarrow \Gamma; v \vdash \pi.P @ D \langle \bar{\pi} \rangle \text{ ispartof } \pi.P @ C \langle \bar{\pi}, \bar{\pi}' \rangle$ | (ispartof Monotonic) |
| $\Gamma; v \vdash (\pi.P @ C \langle \bar{\pi} \rangle * C \text{ classof } \pi) \text{ ** } \pi.P \langle \bar{\pi} \rangle$ | (Known Type) |
| $\Gamma; v \vdash \text{null}.\kappa \langle \bar{\pi} \rangle$ | (Null Receiver) |
| $\Gamma; v \vdash \text{true}$ | (True) |
| $\Gamma; v \vdash \text{false} \text{ ** } F$ | (False) |
| $(\Gamma \vdash e, e' : T \wedge \Gamma, x : T \vdash F : \diamond) \Rightarrow \Gamma; v \vdash (F[e/x] * e == e') \text{ ** } F[e'/x]$ | (Substitutivity) |
| $(\Gamma \models !e_1 \mid !e_2 \mid e') \Rightarrow \Gamma; v \vdash (e_1 * e_2) \text{ ** } e'$ | (Semantic Validity) |
| $\Gamma; v \vdash (\text{PointsTo}(e.f, \pi, e') \& \text{PointsTo}(e.f, \pi', e'')) \text{ assures } e' == e''$ | (Unique Value) |
| $(\Gamma \vdash e : T) \Rightarrow \Gamma; v \vdash (\text{ex } T \alpha) (e == \alpha)$ | (Well-typed) |
| $\Gamma; v \vdash (F \& e) \text{ ** } (F * e)$ | (Copyable) |
| $(\Gamma \vdash \pi : t \langle \bar{\pi}' \rangle \wedge \text{axiom}(t \langle \bar{\pi}' \rangle) = F) \Rightarrow \Gamma; v \vdash F[\pi/\text{this}]$ | (Class) |

TABLE 1. Overview of Axioms

- (Dynamic Type) states 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 the axiom (Open/Close), this allows us to open and close predicates at the receiver's static type. The axiom (ispartof Monotonic) is similar.
- (Known Type) allows one to drop the class modifier C from $\pi.P @ C$ if we know that C is π 's dynamic class.
- Axioms (Null Receiver), (True) and (False) define the semantics of predicates with `null`-receiver, and of `true` and `false`, respectively.
- The (Substitutivity) axiom allows to replace expressions by equal expressions, while (Semantic Validity) lifts semantic validity of boolean expressions to the proof theory.
- (Unique Value) captures the fact that fields point to a unique value. Recall that we write " F assures G " to abbreviate " $F \text{ ** } (F * G)$ " (see Section 2.1).
- (Well-typed) captures that all well-typed closed expressions represent a value (because built-in operations are total).
- (Copyable) expresses copyability of boolean expressions.
- (Class) allows the application of class axioms where $\text{axiom}(t \langle \bar{\pi}' \rangle)$ is the $*$ -conjunction of all class axioms in $t \langle \bar{\pi}' \rangle$ and its supertypes.

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

$$\begin{aligned} \Gamma \vdash \mathcal{E}; \mathcal{H}; s \models F_1, \dots, F_n &\text{ iff } \Gamma \vdash \mathcal{E}; \mathcal{H}; s \models F_1 * \dots * F_n \\ \Gamma \vdash \mathcal{E}; \bar{F} \models G &\text{ iff } (\forall \Gamma, \mathcal{H}, s)(\Gamma \vdash \mathcal{E}; \mathcal{H}; s \models \bar{F} \Rightarrow \Gamma \vdash \mathcal{E}; \mathcal{H}; s \models G) \end{aligned}$$

Now, we can express the proof theory's soundness:

Theorem 2.1 (Soundness of Logical Consequence). *If $\mathcal{F}_{ct}(\mathcal{E}) = \mathcal{E}$ and $(\Gamma; o; \bar{F} \vdash G)$, then $(\Gamma \vdash \mathcal{E}; \bar{F} \models G)$.*

Proof. The proof of the theorem is by induction on $(\Gamma; v; \bar{F} \vdash G)$'s proof tree. The pen and paper proof can be found in [HH08b, §R]. \square

Remark. Note that the receiver parameter o is only used in the assumption, and does not play a role in the semantics of logical consequence. The reason why we included the receiver parameter in the logical consequence judgment is the (Open/Close) axiom. This axiom permits the opening/closing of only those abstract predicates that are defined in the receiver-parameter's class. While limiting the visibility of predicate definitions is not needed for soundness of logical consequence, it is important from a software engineering standpoint, because it provides a well-defined abstraction boundary.

2.4.2. *Hoare Triples.* Next we present Hoare rules to verify programs written in Section 2.1's language. Appendix B of Hurlin's PhD thesis [Hur09] lists the complete collection of Hoare rules, presented here and in the next sections. Hoare triples for *head commands* have the following form: $\Gamma; v \vdash \{F\}hc\{G\}$. Our judgment for *commands* combines typing and Hoare triples: $\Gamma; v \vdash \{F\}c : T\{(U \ \alpha)(G)\}$ where G is the postcondition, α refers to the return value, and T and U are types of the return value (possibly supertypes of the return value's dynamic type). In derivable judgments, it is always the case that $U <: T$.

Here we explain some important rules listed in Figure 3. The rest of the rules are standard and provided in A.6. The field writing (Fld Set) requires the full permission (1) on the object's field and it ensures that the heap has been updated with the value assigned. The rule for field reading (Get) requires a `PointsTo` predicate with *any* permission π . The rule for creating new objects (New) has precondition `true`, because we do not check for out of memory errors. After creating an object, all its fields are writeable: the `l.init` predicate (formally defined in A.1) ***-conjoins the predicates `PointsTo(l.f, 1, df(T))` for all fields Tf in l 's class, i.e., expressing that all fields have their default values. The rule for method calls (Call) is verbose, but standard. Importantly, our system includes the (Frame) rule, which allows to reason locally about methods. To understand this rule, note that $\text{fv}(F)$ is the set of free variables of F and that we write $x \notin F$ to abbreviate $x \notin \text{fv}(F)$. Furthermore, we write `writes(hc)` for the set of read-write variables l that occur freely on the left-hand-side of an assignment in *hc*. (Frame)'s side condition on variables is standard [O'H07, Par05]. Bornat showed how to get rid of this side condition by treating variables as resources [BCY05]. We should stress here that the rule (Consequence) applies *only for* head commands. Therefore, the correctness proof for a method body can never end in an application of a rule of (Consequence). However, it is possible to apply this rule at the caller site and in the proof of the method body at any point before applying the (Val) rule that introduces the outer existential. Notice that since we do not have the *conjunction rule* in our rule set, we do not need the preciseness condition of the resource invariant [GBC11].

$$\begin{array}{c}
 \frac{\Gamma \vdash u, w : U, W \quad W f \in \text{fld}(U)}{\Gamma; v \vdash \{\text{PointsTo}(u.f, 1, W)\} u.f = w \{\text{PointsTo}(u.f, 1, w)\}} \quad (\text{Fld Set}) \\
 \\
 \frac{\Gamma \vdash u, \pi, w : U, \text{perm}, W \quad W f \in \text{fld}(U) \quad W <: \Gamma(\ell)}{\Gamma; v \vdash \{\text{PointsTo}(u.f, \pi, w)\} \ell = u.f \{\text{PointsTo}(u.f, \pi, w) * \ell == w\}} \quad (\text{Get}) \\
 \\
 \frac{C \langle \bar{T} \bar{\alpha} \rangle \in ct \quad \Gamma \vdash \bar{\pi} : \bar{T}[\bar{\pi}/\alpha] \quad C \langle \bar{\pi} \rangle <: \Gamma(\ell)}{\Gamma; v \vdash \{\text{true}\} \ell = \text{new } C \langle \bar{\pi} \rangle \{\ell.\text{init} * C \text{ classof } \ell\}} \quad (\text{New}) \\
 \\
 \frac{\text{mtype}(m, t \langle \bar{\pi} \rangle) = \langle \bar{T} \bar{\alpha} \rangle \text{ requires } G; \text{ ensures } (\alpha') (G'); \\
 U \ m \ (t \langle \bar{\pi} \rangle \iota_0, \bar{W} \ \bar{\iota}) \quad \sigma = (u/\iota_0, \bar{\pi}'/\bar{\alpha}, \bar{w}/\bar{\iota}) \quad \Gamma \vdash u, \bar{\pi}', \bar{w} : t \langle \bar{\pi} \rangle, \bar{T}[\sigma], \bar{W}[\sigma] \quad U[\sigma] <: \Gamma(\ell)}{\Gamma; v \vdash \{u != \text{null} * G[\sigma]\} \ell = u.m(\bar{w}) \{(\text{ex } U[\sigma] \ \alpha') (\alpha' == \ell * G'[\sigma])\}} \quad (\text{Call}) \\
 \\
 \frac{\Gamma; v \vdash \{F\} hc\{G\} \quad \Gamma \vdash H : \diamond \quad \text{fv}(H) \cap \text{writes}(hc) = \emptyset}{\Gamma; v \vdash \{F * H\} hc\{G * H\}} \quad (\text{Frame}) \\
 \\
 \frac{\Gamma; v \vdash \{F'\} hc\{G'\} \quad \Gamma; v; F \vdash F' \quad \Gamma; v; G' \vdash G}{\Gamma; v \vdash \{F\} hc\{G\}} \quad (\text{Consequence}) \\
 \\
 \frac{\Gamma, \alpha : T; v \vdash \{F\} hc\{G\}}{\Gamma; v \vdash \{(\text{ex } T \ \alpha) (F)\} hc\{(\text{ex } T \ \alpha) (G)\}} \quad (\text{Exists}) \\
 \\
 \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 \{(U \ \alpha) (G)\}} \quad (\text{Val})
 \end{array}$$

FIGURE 3. Hoare triples

The following lemma states that Hoare proofs can be normalized for head commands, which is needed in the preservation proof in order to deal with structural rules.

Lemma 2.2 (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 (Consequence) and (Exists) preceded by exactly one application of (Frame), preceded by a rule that is not a structural rule (i.e., a rule different from (Frame), (Consequence) and (Exists)).*

Proof. See [Hur09, Chap. 6]. □

2.5. Verified Programs. To prove soundness of the logic, we need to define the notion of a verified program. We first define judgments for verified interface and classes, which in turn depend on the notions of method and predicate subtyping, and soundness of axioms.

Subtyping. We define method and predicate subtyping. We present the method subtyping rule in its full generality, accounting for logical parameters:

$$\frac{\Gamma, \iota_0 : V_0; \iota_0; \text{true} \vdash (\text{fa } \bar{T}') (\bar{\alpha}) (\text{fa } \bar{V}') (\bar{\iota}) (F' -* \\
 (\text{ex } \bar{W} \ \bar{\alpha}') (F * (\text{fa } U \ \text{result}) (G -* G')))}{\Gamma \vdash \langle \bar{T} \ \bar{\alpha}, \bar{W} \ \bar{\alpha}' \rangle \text{ requires } F; \text{ ensures } G; U \ m (V_0 \ \iota_0, \bar{V} \ \bar{\iota}) \\
 <: \langle \bar{T}' \ \bar{\alpha} \rangle \text{ requires } F'; \text{ ensures } G'; U' \ m (V_0' \ \iota_0, \bar{V}' \ \bar{\iota})}$$

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:

$$\frac{}{\text{pred } P\langle\bar{T} \bar{\alpha}, \bar{T}' \bar{\alpha}'\rangle <: \text{pred } P\langle\bar{T} \bar{\alpha}\rangle}$$

Soundness of Class Axioms. So far **axioms** are used to export useful relations between predicates to clients. A class is **sound** if all its axioms are sound (the lookup function for axioms (**axiom**) is defined in A.1). To prove soundness of axioms, we define a restricted logical consequence judgment that disallows the application of class axioms for proving their soundness, in order to avoid circularities:

$$\vdash' \triangleq \vdash \text{ without class axioms}$$

Verified Interfaces and Classes. Next, we define same sanity conditions on classes and interfaces, which are later used to ensure that we only verify sane programs. Judgment $C\langle\bar{T} \bar{\alpha}\rangle \text{ ext } U$ expresses that: (1) class C does not redeclare inherited fields, and (2) methods and predicates overridden in class C are subtypes of the corresponding methods and predicates implemented in class U . Judgment $I\langle\bar{T} \bar{\alpha}\rangle \text{ type-extends } U$ expresses that: methods and predicates overridden in interface I are subtypes of the corresponding methods and predicates declared in U . Judgment $C\langle\bar{T} \bar{\alpha}\rangle \text{ impl } U$ expresses that: (1) methods and predicates declared in interface U are implemented in C , and (2) methods and predicates implemented in C are subtypes of the corresponding methods and predicates declared in U . These judgments are defined formally in A.6.

Finally, verified methods, verified interfaces and verified classes are defined formally in A.6. Later, when we verify a user-provided program, we will assume that the class table is verified.

Soundness of the Program Logic. We now have all the machinery to define what is a verified program. To do so, 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, $st : \diamond$, which is preserved by the small-step rules of our operational semantics. Our forcing relation \models from Section 2.3.2 assumes formulas without logical variables: we deal with those by substitution, ranged over by $\sigma \in \text{LogVar} \rightarrow \text{SpecVal}$. We let $(\Gamma \vdash \sigma : \Gamma')$ whenever $\text{dom}(\sigma) = \text{dom}(\Gamma')$ and $(\Gamma[\sigma] \vdash \sigma(\alpha) : \Gamma'(\alpha)[\sigma])$ for all α in $\text{dom}(\sigma)$.

Now, we extend the Hoare triple judgment to states:

$$\frac{\Gamma \vdash \sigma : \Gamma' \quad \text{dom}(\Gamma') \cap \text{cfv}(c) = \emptyset \quad \Gamma, \Gamma' \vdash s : \diamond \quad \Gamma[\sigma] \vdash \mathcal{E}; \mathcal{H}; s \models F[\sigma] \quad \Gamma, \Gamma'; r \vdash \{F\}c : \text{void}\{G\}}{\langle h, c, s \rangle : \diamond} \text{ (State)}$$

where $\text{cfv}(c)$ denotes the set of variables that occur freely in an object creation command in c .

The rule for states ensures that there exists an augmented heap \mathcal{H} to satisfy the state's command. The object identifier r in the Hoare triple (last premise) is the current receiver, needed to determine the scope of abstract predicates. Rule (State) enforces the current command to be verified with precondition F and postcondition G . No condition is required on F and G , but note that, by the semantics of Hoare triples, F represents the state's

allocated memory before executing c : if c is not a top level program (i.e., some memory should be allocated for c to execute correctly), choosing a trivial F such as `true` is incorrect. Similarly, G represents the state's memory after executing c .

The judgment $(ct : \diamond)$ is the top-level judgment of our source code verification system, to be read as “class table ct is verified”. Before presenting the preservation theorem, we first give the following lemma, which illustrates how we handle method calls in the preservation proof.

Lemma 2.3. *If $(\Gamma; o \vdash \{ F \} c : T \{ (\text{ex } T \alpha)(G) \})$, $T <: \Gamma(\ell)$ and $(\Gamma; p \vdash \{ (\text{ex } T \alpha)(\alpha == \ell * G) \} c' : U \{ H \})$ then $(\Gamma; o \vdash \{ F \} \ell \leftarrow c; c' : U \{ H \})$.*

Proof. By induction on the structure of c . □

The following theorem shows that the Hoare rules from Section 2.4.2 are sound.

Theorem 2.4 (Preservation). *If $(ct : \diamond)$, $(st : \diamond)$ and $st \rightarrow_{ct} st'$, then $(st' : \diamond)$.*

Proof. In order to deal with structural rules we need Lemma 2.2 in the preservation proof. Based on the assumptions and Lemma 2.2 there is a proof tree for $st : \diamond$ ending in (Consequence), (Exists) or (Frame). Using case analysis on the shape of the head command we prove that there exists a proof tree for $st' : \diamond$ in all the cases (Consequence), (Exists) and (Frame). Details can be found in [Hur09, Chap. 6]. □

From the preservation theorem, we can draw two corollaries: verified programs never dereference `null` and verified programs satisfy partial correctness. To formally state these theorems, we say that a class table ct together with a “main” program c is sound (written $(ct, c) : \diamond$) iff $(ct : \diamond$ and $\text{null}; \emptyset \vdash \{\text{true}\} c : \text{void}\{\text{true}\}$). In the latter judgment, \emptyset represents that the type environment is initially empty, `null` represents that the receiver is initially vacuous, and `true` represents that the top level program has `true` both as a precondition and as a postcondition. Notice that `true` is a correct precondition for top level programs (Java’s `main`), because when a top level program starts to execute, the heap is initially empty.

Lemma 2.5. *If $(ct, c) : \diamond$, then $\text{init}(c) : \diamond$.*

Proof. See [Hur09, Chap. 6]. □

We can now state the first corollary (no `null` dereference) of the preservation theorem. A head command hc is called a *null error* iff it tries to dereference a null pointer, i.e., $hc = (\ell = \text{null}.f)$ or $hc = (\text{null}.f = v)$ or $hc = (\ell = \text{null}.m \langle \bar{\pi} \rangle (\bar{v}))$ for some $\ell, f, v, m, \bar{\pi}, \bar{v}$.

Theorem 2.6 (Verified Programs are Null Error Free). *If $(ct, c) : \diamond$ and $\text{init}(c) \rightarrow_{ct}^* st = \langle h, hc; c', s \rangle$, then hc is not a null error.*

Proof. By $\text{init}(c) : \diamond$ (lemma 2.5) and preservation theorem (Theorem 2.4), we have $st : \diamond$. Suppose by contradiction that hc is a null error. An inspection of the last rules of $(st : \diamond)$ ’s derivation reveals that there must be an environment, predicate environment, augmented heap, stack and value such that the result is a null error. But by definition of \models this is not possible (details in [Hur09, Chap. 6]). □

To state the second corollary of the preservation theorem, we extend head commands with *specification commands*. 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:

$$\begin{aligned} hc \in \text{HeadCmd} & ::= \dots \mid sc \mid \dots \\ sc \in \text{SpecCmd} & ::= \text{assert}(F) \end{aligned}$$

We update Section 2.2's operational semantics to deal with specification commands. Operationally, specification commands are no-ops:

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

$$\begin{array}{c} \text{(Red No Op)} \\ \dots \quad \langle h, sc; c, s \rangle \rightarrow \langle h, c, s \rangle \quad \dots \end{array}$$

Now, we can state the partial correctness theorem. It expresses that if a verified program contains a specification command $\text{assert}(F)$, then F holds whenever the assertion is reached at runtime:

Theorem 2.7 (Partial Correctness).

If $(ct, c) : \diamond$ and $\text{init}(c) \rightarrow_{ct}^* st = \langle h, \text{assert}(F); c, s \rangle$, then $(\Gamma \vdash \mathcal{E}; (h, \mathcal{P}); s \models F[\sigma])$ for some $\Gamma, \mathcal{E} = \mathcal{F}_{ct}(\mathcal{E}), \mathcal{P}$ and $\sigma \in \text{LogVar} \rightarrow \text{SpecVal}$.

Proof. By $\text{init}(c) : \diamond$ (lemma 2.5) and preservation theorem (Theorem 2.4), we know that $st : \diamond$. An inspection of the last rule of $(st : \diamond)$'s derivation reveals that there must be $\Gamma, \mathcal{E} = \mathcal{F}_{ct}(\mathcal{E}), \mathcal{H}, \sigma \in \text{LogVar} \rightarrow \text{SpecVal}$ such that $(\Gamma \vdash \mathcal{E}; (h, \mathcal{P}); s \models F[\sigma])$. \square

2.6. Example: Sequential Mergesort. To show how the verification system works, we specify a (sequential) implementation of mergesort. In the next section, when we add multithreading, we extend this example to parallel mergesort and we verify the parallel implementation w.r.t its specification.

Since our model language has no arrays, we use linked lists. For simplicity, we use integers as values. Alternatively, as in the Java library, values could be objects that implement the `Comparable` interface. Our example contains two classes: `List` and `MergeSort`, defined³ and specified in Figures 4, 5, 6, and 7.

Class List. Figure 4 contains the implementation of class `List`. This class has three methods): method `append` adds a value to the tail of the list; method `concatenate(1, i)` concatenates the i -th first elements of list `1` to the receiver list; and method `get` returns the sub-tail of the receiver starting at the i -th element. Note that these methods use iteration in different ways. In method `append`'s loop, iteration is used to reach the tail of the receiver list, while in method `concatenate`'s second loop, iteration is used to reach elements *up to a certain length* of list `1`. This means that, in the first case, the executing method should have permission to access the whole list, while in the second case, it suffices to have access to the list up to a certain length. To capture this, class `List` defines two state predicates (see Figure 5): (1) `state<n, p, q>` gives access to the first n elements of the receiver list with permissions p on the field `next` and q on the field `val`; and (2) `state<n, 1, p, q>` additionally requires the successor of the n -th element to point to list `1`. Both predicates ensure that the receiver list is at least of length n , because of the test for non-nullness on the next element (`1b!=null`). As a consequence, predicate `state<n, null, p, q>` represents a list of *exact length* n .

³For clarity of presentation, these classes are written using a more flexible language than our formal language. E.g. we allow reading of fields in conditionals and write chains of fields dereferencing.

```

class List extends Object{
  int val; List next;

  void init(val v){ val = v; }

  void append(int v){
    List rec; rec = this;
    while(rec.next!=null){ rec = rec.next; }
    List novel = new List; novel.init(v); rec.next = novel;
  }

  void concatenate(List l,int i){
    List rec; rec = this;
    while(rec.next!=null){ rec = rec.next; }
    while(i>0){ List node = new List; node.init(l.val);
      rec.next = node; l = l.next; rec = rec.next; i = i-1; }
  }

  List get(int i){
    List res;
    if(i==0) res = this;
    if(i > 0) res = next.get(i-1);
    res;
  }
}

```

FIGURE 4. Implementation of class List

```

class List extends Object{
  public pred state<nat n,perm p, perm q> = (n==0 -* True) *
    (n==1 -* [ex List l. PointsTo(next,p,l) * Perm(val,q)]) *
    (n>1 -* [ex List lb. PointsTo(next,p,lb) * Perm(val,q) *
    lb!=null * lb.state<n-1,p,q>]);
  public pred state<nat n,List l, perm p, perm q> = (n==0 -* True) *
    (n==1 -* [PointsTo(next,p,l) * Perm(val,q)]) *
    (n>1 -* [ex List lb. PointsTo(next,p,lb) * Perm(val,q) *
    lb!=null * lb.state<n-1,l,p,q>]);
}

```

FIGURE 5. List state predicates

Finally, Figure 6) provides the method specifications for the methods in class `List`. We should note here that in the specifications provided for the methods, the binders for logical variables are considered implicit. Method `init`'s postcondition refers explicitly to the `List` class. This might look like breaking the abstraction provided by subtyping. However, because method `init` is meant to be called right after object creation (`new List`), `init`'s postcondition can be converted into a form that does not mention the `List` class.

```

class List extends Object{
  requires init; ensures state@List<1,null,1,1>;
  List init(val v)
  requires state<i,null,1,q> * i>0; ensures state<i+1,null,1,q>;
  void append(int v)
  requires state<j,null,1,q> * j>0 * l.state<k,1,q> * k>=i;
  ensures state<j+i,null,1,q> * l.state<k,1,q>;
  void concatenate(List l,int i)
  requires state<j,p,q> * j>=i * i>=0;
  ensures state<i,result,p,q> * result.state<j-i,p,q>;
  List get(int i)
}

```

FIGURE 6. Method contracts of class `List`

E.g. after calling $l = \text{new List}$ and $l.\text{init}()$, the caller knows that `List` is l 's dynamic class (recall that `(New)`'s postcondition includes an `classof` predicate) and can therefore convert the access ticket $l.\text{state}@List<1,null,1,1>$ to $l.\text{state}<1,null,1,1>$ (using axiom `(Known Type)`). Because they are standard, we do not detail the proofs of the methods in class `List`.

Class MergeSort. Figure 7 present the mergesort algorithm. Class `MergeSort` has two fields: a pointer to the list to be inspected, and an integer indicating how many nodes to inspect. The algorithm itself is implemented by methods `sort` and `merge`. For space reasons, we omit the full implementation, as it is standard: method `sort` distinguishes three cases: (i) if there is only one node to inspect, nothing is done; (ii) if there are only two nodes to inspect, the value of the two nodes are compared and swapped if necessary; and (iii) if the list's length is greater than 2, two recursive calls are made to sort the left and the right parts of the list. The next section will present both the implementation and the proof outline of the parallel mergesort algorithm in more detail.

We have proved that mergesort is memory safe (references point to valid memory locations) and that the length of the sorted list is the same as the input list's length. We do not prove, however, that sorting is actually performed. This would require heavier machinery, because we would have to include mathematical lists in our specification language.

Instances of class `MergeSort` are parameterized by the number of nodes they have to inspect. This is required to show that the input list's length is preserved by the algorithm after the two recursive calls in method `sort()`.

In the proof (and also in the proof of the parallel version presented in the next section) we use two special-purpose axioms. Axiom `(Split)` states that a list of length n can be split into a list of length $m1$ and a list of length $m2$ if (1) $m1+m2==n$ and (2) $m1$'s tail points to $m2$'s head. It can be proved by induction over n . Axiom `(Forget-tail)` relates the two versions of predicate `state`. This allows - for example - to obtain the access ticket `state<1,1,1>` after a call to `init` (in combination with axiom `(Known Type)`).


```

class MergeSort<int length> extends Object{
  List list; int num;

  pred state = PointsTo(list,1,1) * PointsTo(num,1,n) *
  l!=null * n >= 1 * n==length * l.state<length,1,1>;

  requires init * l.state<length,1,1> * i>=1 * i==length * l!=null;
  ensures state@MergeSort;
  init(List l, int i){ list = l; num = i; }

  requires state; ensures result.state<length,1,1>;
  List sort(){ /* uses merge, sorts the elements */ }

  requires ll.state<lenleft,1,1> * rl.state<lenright,1,1> *
  lenleft+lenright==length;
  ensures result.state<length,1,1>;
  List merge(List ll,int lenleft,List rl,int lenright){ ... }
}

```

FIGURE 7. Specification of sequential mergesort algorithm

$$\begin{array}{l}
(m1+m2==n * \text{state}\langle n,p,q\rangle) * - * \\
\quad (\text{ex List } l. \text{ state}\langle m1,l,p,q\rangle * l.\text{state}\langle m2,p,q\rangle) \quad (\text{Split}) \\
\text{state}\langle n,l,p,q\rangle - * \text{state}\langle n,p,q\rangle \quad (\text{Forget-tail})
\end{array}$$

3. SEPARATION LOGIC FOR DYNAMIC THREADS

This section extends Section 2’s language with threads with fork and join primitives, à la Java. The assertion language and verification rules are extended to deal with these primitives. The rules support permissions transfer between threads upon thread creation and termination. The resulting program logic is sound, and its use is illustrated on two examples: a parallel implementation of the mergesort algorithm and an implementation of a typical signal-processing pattern.

Convention: In formal material, we use grey background to highlight what are the changes compared to previous sections.

3.1. A Java-like Language with Fork/Join.

Syntax. First, we extend the syntax of Section 2.1’s language with fork and join primitives. We assume that class tables always contain the declaration of class `Thread`, where class `Thread` contains methods `fork`, `join`, and `run`:

```

class Thread extends Object{
  final void fork();
  final void join();
  void run() { null }
}

```

As in Java, the methods `fork` and `join` are assumed to be implemented natively and their behavior is specified by the operational semantics as follows: `o.fork()` creates a new thread, whose thread identifier is `o`, and executes `o.run()` in this thread. Method `fork` should not be called more than once on `o`. Any subsequent call results in blocking of the calling thread. A call `o.join()` blocks until thread `o` has terminated. The `run`-method is meant to be overridden, while methods `join` and `fork` cannot be overridden (as indicated by the `final` modifiers). In Java, `fork` and `join` are not final, because in combination with super calls, this is useful for custom `Thread` classes. However, we leave the study of overrideable `fork` and `join` methods together with super calls as future work.

Runtime Structures. In Section 2.2, our operational semantics \rightarrow_{ct} is defined to operate on states consisting of a heap, a command, and a stack. To account for multiple threads, states are modified to contain a heap and a *thread pool*. A thread pool maps object identifiers (representing `Thread` objects) to threads. Threads consist of a thread-local stack s and a continuation c . For better readability, we write “ s in c ” for threads $t = (s, c)$, and “ o_1 is $t_1 \mid \dots \mid o_n$ is t_n ” for thread pools $ts = \{o_1 \mapsto t_1, \dots, o_n \mapsto t_n\}$:

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

Initialization. The definition of the initial state of a program is extended to account for multiple threads. Below, `main` is some distinguished object identifier for the main thread. The main thread has an empty set of fields (hence the first \emptyset), and its stack is initially empty (hence the second \emptyset):

$$\text{init}(c) = \langle \{\text{main} \mapsto (\text{Thread}, \emptyset)\}, \text{main is } (\emptyset \text{ in } c) \rangle$$

Semantics. The *operational semantics* defined in Section 2.2 is straightforwardly modified to deal with multiple threads. In each case, *one* thread proceeds, while the other threads remain untouched. In addition, to model `fork` and `join`, we change the reduction step (Red Call) to model that it does not apply to `fork` and `join`. Instead, `fork` and `join` are modeled by two new reductions steps ((Red Fork) and (Red Join)):

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

...

(Red Call) $m \notin \{\text{fork}, \text{join}\}$

$$\begin{aligned} h(o)_1 = C \langle \bar{\pi} \rangle \quad \text{mbody}(m, C \langle \bar{\pi} \rangle) = (v_0; \bar{v}).c_m \quad c' = c_m[o/v_0, \bar{v}/\bar{v}] \\ \langle h, ts \mid p \text{ is } (s \text{ in } \ell = o.m(\bar{v}); c) \rangle \rightarrow \langle h, ts \mid p \text{ is } (s \text{ in } \ell \leftarrow c'; c) \rangle \end{aligned}$$

(Red Fork) $h(o)_1 = C \langle \bar{\pi} \rangle \quad o \notin (\text{dom}(ts) \cup \{p\})$

$$\begin{aligned} \text{mbody}(\text{run}, C \langle \bar{\pi} \rangle) = (v).c_r \quad c_o = c_r[o/v] \\ \langle h, ts \mid p \text{ is } (s \text{ in } \ell = o.\text{fork}(); c) \rangle \rightarrow \langle h, ts \mid p \text{ is } (s \text{ in } \ell = \text{null}; c) \mid o \text{ is } (\emptyset \text{ in } c_o) \rangle \end{aligned}$$

(Red Join)

$$\begin{aligned} \langle h, ts \mid p \text{ is } (s \text{ in } \ell = o.\text{join}(); c) \mid o \text{ is } (s' \text{ in } v) \rangle \rightarrow \\ \langle h, ts \mid p \text{ is } (s \text{ in } \ell = \text{null}; c) \mid o \text{ is } (s' \text{ in } v) \rangle \end{aligned}$$

...

In (Red Fork), a new thread o is forked. Thread o 's state consists of an empty stack \emptyset and command c_o . Command c_o is the body of method `run` in o 's dynamic type where the formal receiver `this` and the formal class parameters have been substituted by the actual receiver and the actual class parameters. In (Red Join), thread p joins the terminated thread o . Our rule models that `join` finishes when o is terminated, i.e., its current command is reduced to a single return value. However, notice that the semantics blocks on an attempt to join o , if o has not yet been started. This is different from real Java programs.

3.2. Assertion Language for Fork/Join. This section extends the assertion language to deal with `fork` and `join` primitives. To this end, we introduce a `Join` predicate that controls how threads access postconditions of terminated threads. We also introduce `groups`, which are a restricted class of predicates.

3.2.1. The Join predicate. To model `join`'s behavior, we add a new formula `Join(e, π)` to the assertion language. The intuitive meaning of `Join(e, π)` is as follows: it allows one to pick up fraction π of thread e 's postcondition after e terminated. As a specific case, if π is 1, the thread in which the `Join` predicate appears can pick up thread e 's entire postcondition when e terminates. Thus this formula is used (see Section 3.3) to govern exchange of permissions from terminated threads to alive threads:

$$F ::= \dots \mid \text{Join}(e, \pi) \mid \dots$$

Notice that the same approach can be used to model other synchronisation mechanisms where multiple threads can obtain part of the shared resources.

When a new thread is created, a `Join` predicate is emitted for it. To model this, we redefine the `init` predicate (recall that `init` appears in (New)'s postcondition) for subclasses of `Thread` and for other classes. We do that by (1) adding the following clause to the definition of predicate lookup:

$$\text{plkup}(\text{init}, \text{Thread}) = \text{pred init} = \text{Join}(\text{this}, 1) \text{ ext Object}$$

and (2) adding $C \neq \text{Thread}$ as a premise to the original definition (`Plkup init`). Intuitively, when an object o inheriting from `Thread` is created, a `Join($o, 1$)` ticket is issued.

Augmented Heaps. To express the semantics of the `Join` predicate, we need to change our definition of augmented heaps. Recall that, in Section 2.3.1, augmented heaps were pairs of a heap and a permission table of type $\text{Objld} \times \text{Fieldld} \rightarrow [0, 1]$. Now, we modify permission tables so that they have type $\text{Objld} \times (\text{Fieldld} \cup \{\text{join}\}) \rightarrow [0, 1]$. The additional element in the domain of permission tables keeps track of how much a thread can pick up of another thread's postcondition. Obviously, we forbid `join` to be a program's field identifier.

In addition, we add an additional element to augmented heaps; so that they become triples of a heap, a permission table, and a *join table* $\mathcal{J} \in \text{Objld} \rightarrow [0, 1]$. Intuitively, for a thread o , $\mathcal{J}(o)$ keeps track of how much of o 's postcondition has been picked up by other threads: when a thread gets joined, its entry in \mathcal{J} drops. The compatibility and joining operations on join tables are defined as follows:

$$\mathcal{J} \# \mathcal{J}' \text{ iff } \mathcal{J} = \mathcal{J}' \quad \mathcal{J} * \mathcal{J}' \triangleq \mathcal{J}$$

Because $\#$ is equality, join tables are “global”: in the preservation proof, all augmented heaps will have the same join table⁴. As usual, we define a projection operator: $(h, \mathcal{P}, \mathcal{J})_{\text{join}} \triangleq \mathcal{J}$.

Further, we require augmented heaps to satisfy these additional axioms:

- (c) For all $o \notin \text{dom}(h)$ and all f (including `join`), $\mathcal{P}(o, f) = 0$ and $\mathcal{J}(o) = 1$.
- (d) $\forall o. \mathcal{P}(o, \text{join}) \leq \mathcal{J}(o)$.

Axiom (c) ensures that all unallocated objects have minimal permissions, which is needed to prove soundness of the verification rule for allocating new objects. Axiom (d) ensures that a thread will never try to pick up more than is available of a thread’s postcondition.

Semantics. We update the predicate environments with an axiom to ensure that when a thread is joined, its corresponding entry drops in all join tables. The semantics of the `Join` predicate is as follows:

$$\Gamma \vdash (h, \mathcal{P}, \mathcal{J}); s \models \text{Join}(e, \pi) \text{ iff } \llbracket e \rrbracket_s^h = o \text{ and } \llbracket \pi \rrbracket \leq \mathcal{P}(o, \text{join})$$

Axiom. In analogy with the `PointsTo` predicate, we have a split/merge axiom for the `Join` predicate:

$$\Gamma; v \vdash \text{Join}(e, \pi) \text{ ** } (\text{Join}(e, \frac{\pi}{2}) * \text{Join}(e, \frac{\pi}{2})) \quad (\text{Split/Merge Join})$$

3.2.2. *Groups.* In order to ensure that multiple threads can join a terminated thread, we introduce the notion of *groups*. Groups are special predicates, denoted by keyword `group` that satisfy an additional split/merge axiom. Formally, `group` desugars to a predicate and an axiom:

$$\text{group } P \langle \bar{T} \bar{x} \rangle = F \triangleq \begin{array}{l} \text{pred } P \langle \bar{T} \bar{x} \rangle = F; \\ \text{axiom } P \langle \bar{x} \rangle \text{ ** } (P \langle \text{split}(\bar{T}, \bar{x}) \rangle * P \langle \text{split}(\bar{T}, \bar{x}) \rangle) \end{array}$$

where `split` is extended to pairs of type and parameter, so that it splits parameters of type `perm` and leaves other parameters unchanged:

$$\text{split}(T, x) \triangleq \begin{cases} \text{split}(x) & \text{iff } T = \text{perm} \\ x & \text{otherwise} \end{cases}$$

The meaning of the axiom for groups is as follows: (1) splitting (reading `**` from left to right) P ’s parameters splits predicate P and (2) merging (reading `**` from right to left) P ’s parameters merges predicate P .

⁴This suggests that join tables could be avoided all together in augmented heaps. It is unclear, however, if an alternative approach would be cleaner because rules (State), (Cons Pool), and (Thread) would need extra machinery.

3.3. Contracts for Fork and Join. Next, we discuss how the verification logic for the sequential language, presented in Section 2.4.2 is adapted to cater for the multithreaded setting with `fork` and `join` primitives. Since we can specify contracts in the program logic for `fork` and `join` in class `Thread`, we do not need to give new Hoare rules for them. Instead, rules for `fork` and `join` are simply instances of the rule for method call (Mth). The contracts for `fork` and `join` model how permissions to access the heap are exchanged between threads. Intuitively, newly created threads obtain a part of the heap from their parent thread. Dually, when a terminated thread is joined, (a part of) its heap is transferred to the joining thread(s).

Class Thread. In Section 3.1, we introduced class `Thread` but did not give any specifications. Class `Thread` is specified as follows:

```
class Thread extends Object{
  pred preFork = true;
  group postJoin<perm p> = true;
  requires preFork; ensures true;
  final void fork();
  requires Join(this,p); ensures postJoin<p>;
  final void join();
  final requires preFork; ensures postJoin<1>;
  void run() { null }
}
```

Predicates `preFork` and `postJoin` describe the pre- and postcondition of `run`, respectively. Notice that the contracts of `fork`, `join`, and `run` are tightly related: (1) `fork`'s precondition is the same as `run`'s precondition and (2) `run`'s postcondition is the predicate `postJoin<1>` while `join`'s postcondition is `postJoin<p>`. Point (1) models that when a thread is forked, part of the parent thread's state is transferred to the forked thread. Point (2) expresses that threads joining a thread pick up a part of the joined thread's state. The fact that permission `p` appears both as an argument to `Join` and to `postJoin` (in `join`'s contract) models that joining threads pick up that part of the terminated thread's state which is *proportional* to `Join`'s argument. Because one `Join(o,1)` predicate is issued per thread `o`, and this cannot be duplicated, our system enforces that all threads joining `o` together do not pick up more than thread `o`'s postcondition. The signal-processing example below illustrates reasoning about multiple joins.

Notice that defining `postJoin` as a group is needed because `join`'s postcondition (i.e., `postJoin`) is split among several threads, and by declaring it as a group, we make sure that this splitting is sound.

Although method `run` is meant to be overridden, we require that method `run`'s contract cannot be modified in subclasses of `Thread` (as indicated by the `final` modifier). Subclasses are able to redefine `preFork` and `postJoin` and add parameters to customize the specification. In our examples, this proved to be convenient, however we have not investigated consequences of this choice on more intricate examples. Enforcing `run`'s contract to be fixed allows to express that `join`'s postcondition is proportional to the second parameter of `Join`'s predicate in an easy way (because we can assume that `run`'s postcondition is always `postJoin<1>`).

Since `run`'s contract is fixed, `run`'s contract cannot be parameterized by logical parameters. But this is unproblematic; in fact it would be unsound to allow logical parameters for method `run`. As `run`'s pre and postconditions are interpreted in different threads, one cannot guarantee that logical parameters are instantiated in a similar way between callers to `fork` and callers to `join`. Hence, logical parameters have to be forbidden for `run`.

We highlight that method `run` can also be called directly, without forking a new thread. Our system allows such behavior which is used in practice to flexibly control concurrency (cf Java's `Executors` [Mic]).

Alternative Solutions. Alternatively, we could allow arbitrary contracts for `run`, as we did in our earlier AMAST paper [HH08a]. Yet another solution would be to combine (1) our approach of specifying `fork`, `join`, and `run` with predicates in class `Thread` and (2) to use scalar multiplication as a new *constructor* for formulas (i.e., not a derived form) to express that `run`'s postcondition can be split among joiner threads. This solution, however, requires a thorough study, because having scalar multiplication as a new constructor for formulas may raise semantical issues (as studied by Boyland [Boy07]). Finally, as we stated before, our (Red Join) rule is slightly different from the Java behaviour when a thread tries to join a thread which is not in the thread pool. The contracts proposed here for `fork`, `join` and `run` are adapted in [ABD⁺14] for real Java programs.

3.4. Verified Programs. To extend the definition of a verified program to the multi-threaded setting, we have to update Section 2.5's rules for verified programs to account for multiple threads. First, we craft rules for thread pools:

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

For sequential programs, the core rule extended Hoare triple judgments to states. In the multithreaded setting, this is done in two steps: (1) the rule for states ensures that there exists an augmented heap \mathcal{H} to satisfy the thread pool ts , and (2) a rule for individual thread states corresponds to the original state rule for sequential programs (as defined in Section 2.5). The new state rule looks as follows.

$$\frac{h = \mathcal{H}_{\text{hp}} \quad \mathcal{H} \vdash ts : \diamond}{\langle h, ts \rangle : \diamond} \text{ (State)}$$

The rule for individual threads additionally has to model that threads have a fraction of `postJoin<1>` as postcondition. Therefore, we introduce *symbolic binary fractions* that represent numbers of the forms 1 or $\sum_{i=1}^n \text{bit}_i \cdot \frac{1}{2^i}$:

$$\text{bit} \in \{0, 1\} \quad \text{bits} \in \text{Bits} ::= 1 \mid \text{bit}, \text{bits} \quad \text{fr} \in \text{BinFrac} ::= \text{all} \mid \text{fr}() \mid \text{fr}(\text{bits})$$

Intuitively, we use symbolic binary fractions to speak about finite formulas of the form $r.P\langle 1 \rangle * r.P\langle \frac{1}{2} \rangle * r.P\langle \frac{1}{8} \rangle * \dots$. Formally, we define the scalar multiplication $\text{fr} \cdot r.P\langle \pi \rangle$ as

follows:

$$\begin{aligned}
 \text{all} \cdot r.P\langle\pi\rangle &= r.P\langle\pi\rangle \\
 \text{fr}() \cdot r.P\langle\pi\rangle &= \text{true} \\
 \text{fr}(1) \cdot r.P\langle\pi\rangle &= r.P\langle\text{split}(\pi)\rangle \\
 \text{fr}(0, \text{bits}) \cdot r.P\langle\pi\rangle &= \text{fr}(\text{bits}) \cdot r.P\langle\text{split}(\pi)\rangle \\
 \text{fr}(1, \text{bits}) \cdot r.P\langle\pi\rangle &= r.P\langle\text{split}(\pi)\rangle * \text{fr}(\text{bits}) \cdot r.P\langle\text{split}(\pi)\rangle
 \end{aligned}$$

For instance, $\text{fr}(1, 0, 1) \cdot r.P\langle 1 \rangle * * (r.P\langle \frac{1}{2} \rangle * r.P\langle \frac{1}{8} \rangle)$. The map $\llbracket \cdot \rrbracket : \text{BinFrac} \rightarrow \mathbb{Q}$ interprets symbolic binary fractions as concrete rationals:

$$\llbracket \text{all} \rrbracket \triangleq 1 \quad \llbracket \text{fr}() \rrbracket \triangleq 0 \quad \llbracket \text{fr}(1) \rrbracket \triangleq \frac{1}{2}$$

$$\llbracket \text{fr}(0, \text{bits}) \rrbracket \triangleq \frac{1}{2} \llbracket \text{fr}(\text{bits}) \rrbracket \quad \llbracket \text{fr}(1, \text{bits}) \rrbracket \triangleq \frac{1}{2} + \frac{1}{2} \llbracket \text{fr}(\text{bits}) \rrbracket$$

Now, the rule for individual threads is as follows:

$$\frac{\mathcal{H}_{\text{join}}(o) \leq \llbracket \text{fr} \rrbracket \quad \Gamma \vdash \sigma : \Gamma' \quad \Gamma, \Gamma' \vdash s : \diamond \quad \text{cfv}(c) \cap \text{dom}(\Gamma') = \emptyset \quad \Gamma[\sigma] \vdash \mathcal{E}; \mathcal{H}; s \models F[\sigma] \quad \Gamma, \Gamma'; r \vdash \{F\}c : \text{void}\{\text{fr} \cdot o.\text{postJoin}\langle 1 \rangle\}}{\mathcal{H} \vdash o \text{ is } (s \text{ in } c) : \diamond} \quad (\text{Thread})$$

In rule (Thread), fr should be bigger than the thread considered's entry in the global join table (condition $\mathcal{H}_{\text{join}}(o) \leq \llbracket \text{fr} \rrbracket$). This forces joining threads to take back a part of a terminated thread's postcondition which is not larger than the terminated thread's “remaining” postcondition. This follows from the semantics of the **Join** predicate and the semantics of join tables: $\Gamma \vdash (h, \mathcal{P}, \mathcal{J}); s \models \text{Join}(e, \pi)$ holds iff $\llbracket e \rrbracket_s^h = o$ and $\llbracket \pi \rrbracket \leq \mathcal{P}(o, \text{join})$. Moreover, we have that $\mathcal{P}(o, \text{join}) \leq \mathcal{J}(o)$ (see axiom (d) on page 24).

As in Section 2.5, we have shown that the preservation theorem (Theorem 2.4) holds, and we have shown that verified programs satisfy the following properties: null error freeness and partial correctness. To adapt the proof of Theorem 2.4 to our new settings the only change for existing cases is that there is an extra level of indirection between the top level augmented heap and the augmented heap for each thread regarding this fact that states in multithreading settings include a thread pool. Then using this fact that the proof tree for $(st : \diamond)$ ends in an application of (State), preceded by an application of (Cons Pool), preceded by an application of (Thread) we do case analysis for two additional reduction rule (Red Fork) and (Red Join). The other cases remains the same (details in [Hur09, Chap. 6]).

In addition, verified programs are *data race* free. 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 3.1 (Verified Programs are Data Race Free). *If $(ct, c) : \diamond$ and $\text{init}(c) \rightarrow_{ct}^* st = \langle h, ts \mid o_1 \text{ is } (s_1 \text{ in } hc_1; c_1) \mid o_2 \text{ is } (s_2 \text{ in } hc_2; c_2) \rangle$, then (hc_1, hc_2) is not a data race.*

Proof. By $\text{init}(c) : \diamond$ (Lemma 2.5) and the adapted preservation theorem (Theorem 2.4), 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 be augmented heaps $\mathcal{H}, \mathcal{H}'$ and a heap cell $o.f$ such that: $\mathcal{H} \vdash o_1 \text{ is } (s_1 \text{ in } hc_1; c_1) : \diamond$, $\mathcal{H}' \vdash o_2 \text{ is } (s_2 \text{ in } hc_2; c_2) : \diamond$, $\mathcal{H} \# \mathcal{H}'$, $\mathcal{H}_{\text{perm}}(o, f) = 1$ and $\mathcal{H}'_{\text{perm}}(o, f) > 0$. But then $\mathcal{H}_{\text{perm}}(o, f) + \mathcal{H}'_{\text{perm}}(o, f) > 1$, in contradiction to $\mathcal{H} \# \mathcal{H}'$. \square

```

class MergeSort<int length> extends Thread{
  List list; int num;
  void init(List l, int i){ list = l; num = i; }
  void run(){
    if(num == 1){}
    else{ if(num == 2){
      if(list.val > list.next.val){
        int lval = list.val; list.val = list.next.val; list.next.val = lval;
      } else{
        if(num > 2){
          int lenleft; int lenright;
          if(num % 2 == 0){ lenleft = num / 2; lenright = lenleft; }
          else { lenleft = (num - 1) / 2; lenright = lenleft + 1; }
          List tail = list.get(lenleft);
          MergeSort<lenleft> left = new MergeSort;
          left.init(list,lenleft); left.fork();
          MergeSort<lenright> right = new MergeSort;
          right.init(tail,lenright); right.fork();
          left.join(); right.join(); merge(left,right);
        }
      }
    }
  }
  void merge(MergeSort left, MergeSort right){ .../* standard */ }
}

```

FIGURE 8. Implementation of parallel mergesort algorithm

```

class MergeSort<int length> extends Thread{
  pred preFork = PointsTo(list,1,1) * PointsTo(num,1,n) *
  l!=null * n >= 1 * n==length * l.state<length,1,1>;
  group postJoin<perm p> = PointsTo(list,p,1) * PointsTo(num,p,n) *
  l!=null * n >= 1 * n==length * l.state<length,p,1>;

  requires init * l.state<length,1,1> * length>=1 * i==length * l!=null;
  ensures Join(this,1) * preFork@MergeSort;
  void init(List l, int i) {...}

  requires preFork; ensures postJoin<1>;
  void run() {...}

  requires Perm(list,1) * left.postJoin<1> * right.postJoin<1> * nl+nr==length;
  ensures PointsTo(list,1,1) * l.state<length,1,1>;
  void merge(MergeSort<nl> left, MergeSort<nr> right) {...}
}

```

FIGURE 9. Specification of class MergeSort (parallel version)

3.5. Examples of Reasoning. To illustrate the use of verification system in a multi-threaded setting, we discuss two examples here. First we discuss the verification of a parallel implementation of mergesort, concentrating in particular on the changes in specification and verification because of the use of threads. Second we discuss the verification of a typical digital signal-processing algorithm using multiple joins.

Parallel Mergesort. The parallel mergesort algorithm is a perfect example of disjoint parallelism, because the different threads all modify the *same* list simultaneously but in different places. Figure 8 shows the parallel mergesort implementation, spread over the methods `run` and `merge`. It reuses class `List` from Section 2.6. Similar to the sequential implementation, the class has two fields: a pointer to the list to be inspected, and an integer indicating how many nodes to inspect. Again, method `run` distinguishes three cases, however, in the third case, two new threads are created to sort the left and the right parts of the list, and the parent thread waits for the two new threads and `merges` their results. Figure 9 shows the adapted specifications for the parallel version.

Finally, Figure 10 outlines the correctness proof of method `run`. It illustrates how in the recursive case, the two child threads both receive access to *part of* the parent thread’s list. We use the (Split) axiom (defined in Section 2.6) to specify this behaviour in the proof. This requires some arithmetic reasoning, because threads all have access to the same global list, but then we can conclude that each thread’s access is confined to a limited number of nodes in the list.

Data Plotter. Our next example uses a typical pattern of signal-processing applications to demonstrate how we reason about multiple joins to the same thread. The application has four threads: a sampler, filter processes A and B, and a plotter. The sampler collects the raw data and delivers it to the two processors, which process the raw data in parallel, and stores their results in an appropriate field. Finally, the plotter prints all data (raw and processed). What is important for our example is that both processes A and B join the sampler process, to obtain (read) access to the raw data. To store the data, Figure 11 extends class `List` to contain multiple values. The list structure is captured by predicate `mvstate<n,p>`, which defines a list with length `n` and permission `p` on all the fields of each node of the list. This predicate is defined in terms of predicates `state`, `adata`, `bdata` and `plt`. Predicate `state<n,p/4,p>` is inherited from class `List`; it provides permission `p/4` to the links and permission `p` to the field `val` of each node in the list. Similarly, `adata` and `bdata` define permissions to probe the list and access the fields `outa` and `outb`, respectively. Finally the predicate `plt` provides permissions to visit all nodes in the list.

Figures 12 and 13 show the Sampler thread and process A, respectively, with a proof outline for method `run` of process A. This shows how process A exchanges the half join ticket on Sampler for half of the `postJoin` predicate. Process B is not given, as it similar process A. For space reason, we also do not give the code for the plotter: essentially it joins processes A and B to obtain full access to all data.

Figure 14 shows the main application, with a proof outline of the `process` method. Each thread issues a `Join(this,1)` ticket upon initialization. Method `process` splits the ticket emitted by the sampler and passes each half to the processing threads. Additionally, the join-tickets for the processing threads are transferred to the plotter. The method then waits for the plotter to finish, after which it obtains all permissions on the list back again.

```

    (Let F be the abbreviation of PointsTo(list,1,1) * PointsTo(num,1,n))
    { F * l!=null * n >= 1 * l.state<length,1,1> * n==length }
if(num > 2){
  int lenleft; int lenright;
  if(num % 2 == 0){
    lenleft = num / 2; lenright = lenleft;
    ((Split) axiom with m1 == m2 == n/2 == lenleft == lenright)
    { F * n > 2 * n==length *
      l.state<lenleft,r,1,1> * r.state<lenright,1,1> * lenleft+lenright==length }
  } else { lenleft = (num - 1) / 2; lenright = lenleft + 1;
    ((Split) axiom with m1 == (n-1)/2 and m2==[(n-1)/2]+1)
    { F * n > 2 * n==length *
      l.state<lenleft,r,1,1> * r.state<lenright,1,1> * lenleft+lenright==length }
  }
  (In both cases, we have:)
  { F * n > 2 * n==length *
    l.state<lenleft,r,1,1> * r.state<lenright,1,1> * lenleft+lenright==length }
  ((Split) axiom from right to left)
  { F * n > 2 * n==length * l.state<n,1,1> * lenleft+lenright==length }
  (This matches get's precondition, because 1/ n>=lenleft follows from
   lenleft+lenright==length and 2/ lenleft>=0 follows from
   num==length and length>=0 (not shown in this proof outline).)
  List tail = list.get(lenleft);
  (Let G be the abbreviation of n>2 * lenleft+lenright==length * n==length)
  { F * G * l.state<lenleft,tail,1,1> * tail.state<n-lenleft,1,1> }
  (axiom (Forget-tail) and arithmetic (n-lenleft==lenright))
  { F * G * l.state<lenleft,1,1> * tail.state<lenright,1,1> }
  MergeSort<lenleft> left = new MergeSort; left.init(list,lenleft);
  { F * G * tail.state<lenright,1,1> * left.preFork * Join(left,1) }
  left.fork();
  { F * G * tail.state<lenright,1,1> * Join(left,1) }
  MergeSort<lenright> right = new MergeSort; right.init(tail,lenright);
  right.fork();
  { F * G * Join(left,1) * Join(right,1) }
  left.join();
  { F * G * left.postJoin<1> * Join(right,1) }
  right.join();
  { F * G * left.postJoin<1> * right.postJoin<1> }
  (This matches merge's precondition because (1) the type system
   tells us: left : MergeSort<lenleft> and right : MergeSort<lenright>
   (2) F entails Perm(list,1), and
   (3) G entails lenleft+lenright==length)
  merge(left,right);
  { F * G * l.state<length,1,1> }
  (Close)
  { postJoin<1> }

```

FIGURE 10. Correctness proof of method run in class MergeSort

Notice that this enables the main thread to iterate on the whole processing chain (not shown here).

4. SEPARATION LOGIC FOR REENTRANT LOCKS

This section presents verification rules for Java’s reentrant locks. Together with `fork` and `join`, reentrant locks are a crucial feature of Java for multithreaded programs. In particular, locks serve to synchronize threads and to control access to resources.

Reentrant locks can be acquired more than once by the same thread. They are a convenient tool for programmers, but they also require extra machinery in the verification system, because initial acquisitions have to be distinguished from reentrant acquisitions.

After a short background discussion on modeling single-entrant locks in separation logic, we discuss how syntax and semantics are extended to model reentrant locks. We develop appropriate verification rules, and discuss how their soundness can be proven. We finish the section by some examples that illustrate reasoning about re-entrant locks and the wait-notify mechanisms.

4.1. Separation Logic and Single-Entrant Locks. Separation logic for a programming language with locks as a concurrency primitive has been first explored by O’Hearn [O’H07] in which he elegantly adapted an old idea from concurrent programs with shared variables [And91]. Each lock is associated with a *resource invariant* that describes the part of the heap that the lock guards. When a lock is acquired, it lends its resource invariant to the

```

class MVList extends List {
  int outa; int outb;

  public pred adata<nat n, perm p, perm q> = (n==0 -* true) *
    ( n==1 -* [ex List l. PointsTo(next, p, l) * Perm(outa,q) ] ) *
    ( n>=1 -* [ex List l. PointsTo(next, p, l) * Perm(outa,q) ] *
      l!=null * l.adata<n-1,p,q> );

  public pred bdata<nat n, perm p, perm q> = (n==0 -* true) *
    ( n==1 -* [ex List l. PointsTo(next, p, l) * Perm(outb,q) ] ) *
    ( n>=1 -* [ex List l. PointsTo(next, p, l) * Perm(outb,q) ] *
      l!=null * l.bdata<n-1,p,q> );

  public pred plt<nat n, perm p> = (n==0 -* true) *
    ( n==1 -* [ex List l. PointsTo(next, p, l) ] ) *
    ( n>=1 -* [ex List l. PointsTo(next, p, l) ] *
      l!=null * l.plt<n-1,p> );

  public pred mvstate<nat n,perm p> = super.state<n,p/4,p> *
    adata<n,p/4,p> * bdata<n,p/4,p> * plt<n,p/4>;
}

```

FIGURE 11. Multi-valued list extending class `List`

```

class Sampler<int len> extends Thread {
  MVList<len> lst;

  pred preFork = PointsTo(lst,1,1) * l!=null * l.state<len,1/4,1>;
  group postJoin<perm p> = PointsTo(lst,p,1) * l!=null * l.state<len,p/4,p>;

  requires init * Perm(lst,1) * l!=null;
  ensures Join(this,1) * PointsTo(lst,1,1) * l!=null * l.state<len,1/4,1>;
  void init(MVList l) { lst = l; }

  requires preFork; ensures postJoin<1>;
  void run() { sample(); }

  requires lst.state<len, 1/4, 1>; ensures lst.state<len,1/4, 1>;
  void sample() { ... /*(fills raw data fields ( vals ) with samples)*/ }
}

```

FIGURE 12. The Sampler thread

```

class AFilter<int len> extends Thread {
  MVList<len> lst; Sampler<len> sampler;

  pred preFork = PointsTo(lst,1,1) * l!=null * l.adata<len,1/4,1> *
    Perm(sampler,1) * Join(sampler,1/2) ;
  group postJoin<perm p> = Perm(sampler,p) * PointsTo(lst,p,1) * l!=null *
    l.state<len,p/8,p/2> * l.adata<len,p/4,p>;

  requires init * Perm(sampler,1) * Perm(lst,1) * l.adata<len,1/4,1> * l!=null;
  ensures Join(this,1) * lst.adata<len,1/4,1> * len >= 1 * lst!=null;
  void init(MVList l, Sampler s) { lst = l; sampler = s; }

  requires preFork; ensures postJoin<1>;
  void run() {
    { Join(sampler,1/2) * lst.adata<len,1/4,1> }
    sampler.join();
    { lst.state<len,1/8,1/2> * lst.adata<len,1/4,1> }
    processA();
    { lst.state<len,1/8,1/2> * lst.adata<len,1/4,1> }
  }

  requires lst.state<len,p,q> * lst.adata<len,r,1>;
  ensures lst.state<len,p,q> * lst.adata<len,r,1>;
  void processA() { ... /* using raw data computes outa fields. */ }
}

```

FIGURE 13. Processing thread A

```

class Process<int len> extends Object{
  MVList<len> data;

  requires Perm(data,1) * l!=null * l.mvstate<len,1>;
  ensures PointsTo(data,1,l) * l!=null * l.mvstate<len,1>;
  void init(MVList l) { data = l; }

  requires PointsTo(data,1,l) * l !=null * l.mvstate<len,1>;
  ensures PointsTo(data,1,l) * l !=null * l.mvstate<len,1>;
  void process(MVList lst) {
    (Let abbreviate: data.state<len,1/4,1> with S, data.adata<len,1/4,1> with A
     data.bdata<len,1/4,1> with B, data.plt<len,1/4> with P
    { S * A * B * P }
    Sampler<len> smp = new Sampler; smp.init(data);
    { Join(smp,1) * smp.preFork * A * B * P }
    AFilter<len> af = new AFilter; af.init(data, smp);
    { Join(smp,1/2) * Join(af,1) * smp.preFork * af.preFork * B * P }
    BFilter<len> bf = new BFilter; bf.init(data, smp);
    { Join(af,1) * Join(bf,1) * smp.preFork * af.preFork * bf.preFork * P }
    Plotter<len> plt = new Plotter; plt.init(data,af,bf);
    { Join(plt,1) * smp.preFork * af.preFork * bf.preFork * plt.preFork }
    smp.fork(); af.fork(); bf.fork(); plt.fork();
    { Join(plt,1) }
    plt.join();
    { plt.postJoin<1> }
  }
}

```

FIGURE 14. The main process

acquiring thread. Dually, when a lock is released, it takes back its resource invariant from the releasing thread. This is formally expressed by the following Hoare rules:

$$\frac{I \text{ is } x\text{'s resource invariant}}{\{\text{true}\}x.\text{lock()}\{I\}} \quad \frac{I \text{ is } x\text{'s resource invariant}}{\{I\}x.\text{unlock()}\{\text{true}\}}$$

While these rules are sound for single-entrant locks, they are unsound for reentrant locks, because they allow “duplicating” a lock’s resource invariant:

```

{ true }
x.lock(); // I is x's resource invariant
{ I }
x.lock();
{ I*I } ← wrong!

```

4.2. A Java-like Language with Reentrant Locks. To recover soundness in the presence of reentrant locks, we design proof rules that distinguish between initial acquirement and reentrant acquirement of locks. This allows transferring a lock’s resource invariant to

an acquiring thread only at initial acquirement. In contrast to existing work that studies simple "while" languages and "C-like" languages [O'H07, HAN08, GBC⁺07], we also handle reentrancy.

Syntax. First we extend the syntax and the semantics of our Java-like language to model reentrant locks. We extend the list of head commands defined in Section 2.1 as follows:

$$hc \in \text{HeadCmd} ::= \dots \mid v.\text{lock}() \mid v.\text{unlock}() \mid \dots$$

Just as 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 [O'H07], 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.

Therefore, we distinguish between *fresh* locks and *initialized* locks. A fresh lock does not yet guard its resource invariant and thus it is not ready to be acquired yet. An initialized lock, however, is ready to be acquired. Initially, locks are fresh and they might become initialized later (and then will remain initialized). We require programmers to explicitly change the state of locks (from fresh to initialized) with a `commit` command:

$$sc \in \text{SpecCmd} ::= \dots \mid \pi.\text{commit} \mid \dots$$

Operationally, `π .commit` is a no-op; semantically it checks that π is fresh and changes π 's state to initialized. For expressiveness `commit`'s receiver ranges over specification variables, which include both program variables and logical variables (such as class parameters). In real-world Java programs, a possible default would be to add a `commit` command at the end of constructors.

We assume that class tables always contain the following class declaration:

```
class Object {
  pred inv = true;
  final void wait();
  final void notify();
  final void notifyAll();
}
```

The distinguished `inv` predicate assigns to each lock a resource invariant. The definition `true` is a default and objects meant to be used as locks should extend `inv`'s definition in subclasses of `Object` (just like any other abstract predicate). As usual [O'H07], 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.

The methods `wait`, `notify` and `notifyAll` do not have Java implementations, but are implemented natively. To model this, our operational semantics (page 36) specifies their

$$\begin{aligned}
 l \in \text{LockTable} &= \text{ObjId} \rightarrow \{\text{free}\} \uplus (\text{ObjId} \times \mathbb{N}) && \text{(Lock Table)} \\
 st \in \text{State} &= \text{Heap} \times \text{LockTable} \times \text{ThreadPool} && \text{(States)} \\
 \text{init}(c) &= \langle \{\text{main} \mapsto (\text{Thread}, \emptyset)\}, \emptyset, \text{main is } (\emptyset \text{ in } c) \rangle && \text{(Initialization)}
 \end{aligned}$$

FIGURE 15. Run-time structure, states and initialization in presence of locks

behavior explicitly. If $o.\text{wait}()$ is called when object o is locked at reentrancy level n , then o 's lock is released and the executing thread temporarily stops executing. If $o.\text{notify}()$ is called, one thread that is stopped (because this thread called $o.\text{wait}()$ before) resumes and starts competing for o 's lock. When a resumed thread reacquires o 's lock, its previous reentrancy level is restored. The method `notifyAll` performs similar to `notify` except that it resumes all the waiting threads. Since we can specify method contracts for `wait` and `notify`, we do not put them in our set of commands (see Section 4.4). In contrast, `lock`, `unlock`, and `commit` are put in our set of commands, because the Hoare rules for these methods cannot be expressed using the syntax of contracts available to programmers: we need extra expressivity (see Section 4.4).

Runtime Structures and Initialization. To represent locks in the operational semantics, we use a *lock table*. *Lock tables* map objects o to either the symbol `free`, or to the thread object t that currently holds o 's lock and a number that counts how often t currently holds o . Accordingly states and the initial state of a program are extended with a lock table (see Figure 15). Initially, the lock table of a program is empty (hence the second \emptyset).

Operational Semantics. We modify the *operational semantics* defined in Section 3.1 to deal with locks. 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 as special head commands:

$$hc ::= \dots \mid o.\text{waiting}(n) \mid o.\text{resume}(n) \mid \dots$$

Restriction: These clauses must not occur in source programs.

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

Below we list the existing cases of the operational semantics that are slightly modified: (Red New) and (Red Call), and the cases that are added: (Red Lock), (Red Unlock), (Red Wait), (Red Notify), (Red Skip Notify), (Red Notify All), (Red Skip Notify All) and (Red Resume). Except that a lock table is added, most of the existing cases of the operational semantics are left untouched.

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

...

$$\begin{aligned}
 \text{(Red New)} \quad & o \notin \text{dom}(h) \quad h' = h[o \mapsto (C\langle\bar{\pi}\rangle, \text{initStore}(C\langle\bar{\pi}\rangle))] \\
 & s' = s[\ell \mapsto o] \quad l' = l[o \mapsto \text{free}] \\
 & \langle h, l, ts \mid p \text{ is } (s \text{ in } \ell = \text{new } C\langle\bar{\pi}\rangle; c) \rangle \rightarrow \langle h', l', ts \mid p \text{ is } (s' \text{ in } c) \rangle
 \end{aligned}$$

- (Red Call) $m \notin \{\text{fork, join, wait, notify, notifyAll}\}$
 $h(o)_1 = C\langle\bar{\pi}\rangle \quad \text{mbody}(m, C\langle\bar{\pi}\rangle) = (v_0, \bar{v}).c_m \quad c' = c_m[o/v_0, \bar{v}/\bar{v}]$
 $\langle h, l, ts \mid p \text{ is } (s \text{ in } \ell = o.m(\bar{v}); c) \rangle \rightarrow \langle h, l, ts \mid p \text{ is } (s \text{ in } \ell \leftarrow c'; c) \rangle$
- (Red Lock) $(l(o) = \text{free}, l' = l[o \mapsto (p, 1)])$ or $(l(o) = (p, n), l' = l[o \mapsto (p, n + 1)])$
 $\langle h, l, ts \mid p \text{ is } (s \text{ in } o.\text{lock}(); c) \rangle \rightarrow \langle h, l', ts \mid p \text{ is } (s \text{ in } c) \rangle$
- (Red Unlock) $l(o) = (p, n) \quad n = 1 \Rightarrow l' = l[o \mapsto \text{free}]$
 $n > 1 \Rightarrow l' = l[o \mapsto (p, n - 1)]$
 $\langle h, l, ts \mid p \text{ is } (s \text{ in } o.\text{unlock}(); c) \rangle \rightarrow \langle h, l', ts \mid p \text{ is } (s \text{ in } c) \rangle$
- (Red Wait) $l(o) = (p, n) \quad l' = l[o \mapsto \text{free}]$
 $\langle h, l, ts \mid p \text{ is } (s \text{ in } \ell = o.\text{wait}(); c) \rangle$
 $\rightarrow \langle h, l', ts \mid p \text{ is } (s \text{ in } o.\text{waiting}(n); o.\text{resume}(n); c) \rangle$
- (Red Notify) $l(o) = (p, n)$
 $\langle h, l, ts \mid p \text{ is } (s \text{ in } \ell = o.\text{notify}(); c) \mid q \text{ is } (s' \text{ in } o.\text{waiting}(n'); c') \rangle$
 $\rightarrow \langle h, l, ts \mid p \text{ is } (s \text{ in } c) \mid q \text{ is } (s' \text{ in } c') \rangle$
- (Red Notify All) $l(o) = (p, n)$
 $\langle h, l, ts \mid p \text{ is } (s \text{ in } \ell = o.\text{notifyAll}(); c) \mid q \text{ is } (s' \text{ in } o.\text{waiting}(n'); c') \rangle$
 $\rightarrow \langle h, l, ts \mid p \text{ is } (s \text{ in } \ell = o.\text{notifyAll}(); c) \mid q \text{ is } (s' \text{ in } c') \rangle$
- (Red Skip Notify) $l(o) = (p, n)$
 $\langle h, l, ts \mid p \text{ is } (s \text{ in } \ell = o.\text{notify}(); c) \rangle \rightarrow \langle h, l, ts \mid p \text{ is } (s \text{ in } c) \rangle$
- (Red Skip Notify All) $l(o) = (p, n)$
 $\langle h, l, ts \mid p \text{ is } (s \text{ in } \ell = o.\text{notifyAll}(); c) \rangle \rightarrow \langle h, l, ts \mid p \text{ is } (s \text{ in } c) \rangle$
- (Red Resume) $l(o) = \text{free} \quad l' = l[o \mapsto (p, n)]$
 $\langle h, l, ts \mid p \text{ is } (s \text{ in } o.\text{resume}(n); c) \rangle \rightarrow \langle h, l', ts \mid p \text{ is } (s \text{ in } c) \rangle$
- ...

Rule (Red Lock) distinguishes two cases: (1) lock o is acquired for the first time ($l(o) = \text{free}$) and (2) lock o is acquired reentrantly ($l(o) = (p, n)$). Similarly, rule (Red Unlock) distinguishes two cases: (1) lock o 's reentrancy level decreases but o remains acquired ($l(o) = (p, n)$ and $n > 1$) and (2) lock o is released ($l(o) = (p, 1)$). Rule (Red Wait) fires only if the thread considered previously acquired `wait`'s receiver. In this case, `wait`'s receiver is released and the thread enters the `waiting` state. The thread's reentrancy level is stored in `waiting`'s argument.

Like rule (Red Wait), the rules (Red Notify), (Red Notify All), (Red Skip Notify) and (Red Skip Notify All) fire only if the thread considered previously acquired `notifyAll`'s receiver. The rules (Red Notify) and (Red Notify All) fires if there *exists* at least one thread waiting on `notify`'s receiver. In case of (Red Notify), one of the waiting threads (arbitrarily) is resumed, while (Red Notify All) awakes all the waiting threads. If there is no thread waiting on `notify`'s receiver, based on the calling command, either (Red Skip Notify) or (Red Skip Notify All) fires. In this case, the call to `notify` and `notifyAll` has no effect on other threads. In Java, if `o.wait()`, `o.notify()` and `o.notifyAll()` are called by a thread that does not hold o , an `IllegalMonitorState` exception is raised. In our semantics, this is modeled by being stuck. In Section 4.4, we will give preconditions for `wait` and `notify` that ensure that verified programs would never throw an `IllegalMonitorState` exception (got stuck in our model). Rule (Red Resume) resumes a thread that previously waited on some lock and restores the reentrancy level.

4.3. Separation Logic for Reentrant Locks. In this section, we describe the new formulas that we add to the specification language of Section 3.2.

As explained earlier, a proof system for reentrant locks must keep track of the locks that the current thread holds. To this end, we enrich our specification language:

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

It is convenient to allow using objects as singleton locksets (rather than introducing explicit syntax for converting from objects to singleton locksets). We classify the new formulas into *copyable* and *non-copyable* ones. Copyable formulas represent *persistent state properties* (i.e., properties that hold forever, once established), whereas non-copyable formulas represent *transient state properties* (i.e., properties that hold temporarily). For copyable F , we postulate the axiom $(G \& F) \text{-*} (G * F)$, whereas for non-copyable formulas we postulate no such axiom. Note that this axiom implies $F \text{-*} (F * F)$, hence the term “copyable”. The new specification values and formulas have the following intuitive meaning:

- **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*)

Initialization. When verifying the body of `Thread.run()`, we assume `Lockset(nil)` as a precondition. As explained before, resource invariants must be initialized before the associated locks can be acquired. To keep track of the state of locks in our verification system, we introduce two more formulas:

$$\begin{aligned} F \in \text{Formula} & ::= \dots \mid e.\text{fresh} \mid e.\text{initialized} \mid \dots \\ \textit{Restriction: } & e.\text{initialized} \text{ must not occur in negative positions.} \end{aligned}$$

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

Because `$e.\text{initialized}$` is copyable, `initialized` formulas can “spread” to all threads, allowing all threads to try to acquire locks (`initialized` will be a precondition for (initial) lock acquirement; see Section 4.4).

Types. We add a type to represent locksets and we postulate `Object <: lockset`:

$$T ::= \dots \mid \text{lockset} \mid \dots$$

Because we allow arbitrary specification values (including locksets) as type parameters, we consider 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:

Equivalence of Specification Values: $\pi \simeq \pi$

| |
|---|
| $\text{nil} \cdot \pi \simeq \pi \quad \pi \cdot \pi' \simeq \pi' \cdot \pi \quad (\pi \cdot \pi') \cdot \pi'' \simeq \pi \cdot (\pi' \cdot \pi'')$ |
|---|

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

Augmented heaps. To express the semantics of the new formulas, we need to extend augmented heaps with three new components. From now on, augmented heaps are 6-tuples of a heap, a permission table, a join table, an *abstract lock table* $\mathcal{L} \in \text{ObjId} \rightarrow \text{Bag}(\text{ObjId})$, a *fresh set* $\mathcal{F} \subseteq \text{ObjId}$, and an *initialized set* $\mathcal{I} \subseteq \text{ObjId}$.

Abstract lock tables map thread identifiers to locksets. Just as permission tables are an abstraction of heaps, abstract lock tables are an abstraction of lock tables. The compatibility relation captures that distinct threads cannot hold the same lock (we use \sqcap to denote bag intersection, \sqcup for bag union, and $[]$ for the empty bag):

$$\mathcal{L} \# \mathcal{L}' \text{ iff } \begin{cases} \text{dom}(\mathcal{L}) \cap \text{dom}(\mathcal{L}') = \emptyset \\ (\forall o \in \text{dom}(\mathcal{L}), p \in \text{dom}(\mathcal{L}')) (\mathcal{L}(o) \sqcap \mathcal{L}'(p) = []) \end{cases} \quad ; \text{ and } \quad \mathcal{L} * \mathcal{L}' \triangleq \mathcal{L} \cup \mathcal{L}'$$

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

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

We require augmented heaps to satisfy the following axioms (in addition to Section 3.2's axioms):

(e) $\mathcal{F} \cap \mathcal{I} = \emptyset$.

(f) If $o \in \mathcal{L}(p)$ then $o \in \mathcal{I}$.

Axiom (e) ensures that an object can never be both fresh *and* initialized. Axiom (f) ensures that locked objects are initialized.

As usual, we define projection operators:

$$(h, \mathcal{P}, \mathcal{J}, \mathcal{L}, \mathcal{F}, \mathcal{I})_{\text{lock}} \triangleq \mathcal{L} \quad (h, \mathcal{P}, \mathcal{J}, \mathcal{L}, \mathcal{F}, \mathcal{I})_{\text{fresh}} \triangleq \mathcal{F} \quad (h, \mathcal{P}, \mathcal{J}, \mathcal{L}, \mathcal{F}, \mathcal{I})_{\text{init}} \triangleq \mathcal{I}$$

Semantics of Values. Before defining the semantics of formulas, we extend the semantics of values to locksets. Recall that SemVal is the set of semantic values (defined in Section 2.3.2). Formerly, $\text{SemVal} = \{\text{null}\} \cup \text{ObjId} \cup \text{Int} \cup \text{Bool} \cup (0, 1]$. We extend this set to include semantic domains for locksets:

$$\mu \in \text{SemVal} \triangleq (\{\text{null}\} \cup \text{ObjId} \cup \text{Int} \cup \text{Bool} \cup (0, 1] \cup \text{Bag}(\text{ObjId})) / \equiv$$

where \equiv is the least equivalence relation on SemVal 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.

Let WellTypCISpecVal be the set of well-typed, specification values:

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

To define the semantics of well-typed, open specification values, we simply define the semantics of the two new specification values:

$$[[\cdot]] : \text{WellTypCISpecVal} \rightarrow \text{SemVal} \quad [[\text{nil}]] \triangleq [] \quad [[\pi \cdot \pi']] \triangleq [[\pi]] \sqcup [[\pi']]$$

| | |
|---|--------------|
| $\Gamma; v \vdash !(\text{nil contains } e)$ | (Member Nil) |
| $\Gamma; v \vdash (\pi \cdot \pi') \text{ contains } e \text{ ** } (\pi \text{ contains } e \mid \pi' \text{ contains } e)$ | (Member Rec) |
| $\pi \simeq \pi' \Rightarrow \Gamma; v \vdash \pi == \pi'$ | (Eq Bag) |
| $\Gamma; v \vdash \pi == \pi$ | (Eq Refl) |
| $\Gamma; v \vdash \pi == \pi' \Rightarrow \Gamma; v \vdash \pi' == \pi$ | (Eq Sym) |
| $\Gamma; v \vdash \pi == \pi' \ \& \ \pi' == \pi'' \Rightarrow \Gamma; v \vdash \pi == \pi''$ | (Eq Trans) |
| $G \in \{e, \pi \text{ contains } e, e.\text{initialized}\}$ | |
| \Downarrow | (Copyable) |
| $\Gamma; v \vdash (F \ \& \ G) \text{ ** } (F \ * \ G)$ | |

TABLE 2. Axioms to reasons about bags and copyable formulas

Semantics of Formulas. We now state the semantics of formulas introduced to deal with reentrant locks:

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

These clauses are self-explanatory, except perhaps the existential quantification in the clause for $\text{Lockset}(\pi)$. Intuitively, this clause says that there exists a thread identifier o in \mathcal{L} 's domain such that π denotes the current lockset associated with o . When we interpret an assertion for a single thread, we restrict the models to the ones where \mathcal{L} only contains a single entry for the current thread. Hence, the existential can only choose the current thread id. This restriction is in the (Thread) rule on page 43.

Axioms. Table 2 lists the new axioms that can be used as an extension to the logical consequence judgement (similar to the axioms in Table 1. These axiomatize bag membership ((Member Nil) and (Member Rec)); bag equality ((Eq Bag)); copyability and equality between specification values ((Eq Refl), (Eq Sym), (Eq Trans)). Axiom (Copyable) updates Section 2.4.1's (Copyable) axiom about copyability of formulas. It is straightforward to extend Theorem 2.1 to these new axioms.

4.4. Hoare Triples. In this section, we modify the Hoare triple (New) for allocating new objects and we present Hoare triples for the new commands of our language.

We modify rule (New) so that it emits the **fresh** predicate in its postcondition:

$$\frac{C \langle \bar{T} \bar{\alpha} \rangle \in ct \quad \Gamma \vdash \bar{\pi} : \bar{T}[\bar{\pi}/\alpha] \quad C \langle \bar{\pi} \rangle \prec : \Gamma(\ell)}{\Gamma; v \vdash \ell = \text{new } C \langle \bar{\pi} \rangle \quad \{ \ell.\text{init} * C \text{ classof } \ell * \textcircled{\otimes}_{\Gamma(u) \prec : \text{Object}} \ell != u * \ell.\text{fresh} \}} \quad \text{(New)}$$

In addition to the usual **init** and **classof** predicates, (New)'s 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 avoids explicit reasoning about the absence of aliasing. Unfortunately, we need this kind of reasoning when

establishing the precondition for the rule (Lock) below, which requires that the lock is *not* already held by the current thread.

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

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

Intuitively, the fact that $\pi.\text{inv}$ appears in (Commit)'s precondition but does not appear in (Commit)'s postcondition indicates that after being committed, lock π *guards* its resource invariant: the resource invariant $\pi.\text{inv}$ has been given to lock π and $\pi.\text{inv}$ is not available anymore to the executing thread. Furthermore, because $\pi.\text{fresh}$ only holds if $\pi \neq \text{null}$, this rule ensures that only non-null locks can become initialized.

The precondition of rule (Commit) ensures that monitor invariants cannot mention **Lockset** predicates as it mentions both a **Lockset** predicate and the lock's monitor invariant inv . This follows from the semantics of the **Lockset** predicate and the semantics of the $*$ operator: two **Lockset** predicates cannot be $*$ -conjoined. This is important because **Lockset** predicates are interpreted w.r.t. the current thread.

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.

First, we present the two rules for locking:

$$\frac{\Gamma \vdash u, \pi : \text{Object}, \text{lockset}}{\Gamma; v \vdash \{\text{Lockset}(\pi) * !(\pi \text{ contains } u) * u.\text{initialized}\} \quad u.\text{lock()} \quad \{\text{Lockset}(u \cdot \pi) * u.\text{inv}\}} \quad (\text{Lock})$$

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

The rule (Lock) applies when lock u is acquired non-reentrantly, as expressed by the precondition $\text{Lockset}(\pi) * !(\pi \text{ contains } u)$. The precondition $u.\text{initialized}$ makes sure that (1) threads only acquire locks whose resource invariant is initialized, and (2) no null-error can happen (because initialized values are non-null). The postcondition adds u to the current thread's lockset, and assumes u 's resource invariant. The resource invariant obtained is $u.\text{inv}$ (without $@$ selector).

Proving (Lock)'s precondition requires reasoning about aliases because one has to prove $!(\pi \text{ contains } u)$. In practice, this assertion is proven by showing that u is different from all elements of lockset π . Such a reasoning is a form of alias analysis. On one hand this is unfortunate, because separation logic's power comes from the fact that it does not need to reason about aliases. On the other hand, this seems unavoidable. Whether this is problematic in practice needs to be investigated on large case studies. In Section 4.6, the lock coupling example illustrates how ownership can be used as a possible solution to the problem.

The rule (Re-Lock) applies when a lock is acquired reentrantly. The precondition of (Re-Lock), contrary to (Lock), does not require $u.\text{initialized}$, because this follows from $\text{Lockset}(u \cdot \pi)$ (locksets contain only initialized values).

Based on rule (Re-Lock) one expects the lock’s resource invariant to hold. To have a more accurate feedback, in practice, one may define a derived rule like (Re-Lock-Accurate) to enforce the existence of the resource invariant:

$$\frac{\Gamma \vdash u, \pi : \text{Object}, \text{lockset}}{\Gamma; v \vdash \frac{\{\text{Lockset}(u \cdot \pi) * u.\text{inv}\}}{u.\text{lock()}} \{\text{Lockset}(u \cdot u \cdot \pi) * u.\text{inv}\}} \quad (\text{Re-Lock-Accurate})$$

Next, we present the two rules for unlocking:

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

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

The rule (Re-Unlock) applies when u ’s current reentrancy level is at least 2 and (Unlock) applies when u ’s resource invariant holds in the precondition.

Other Hoare Rules that Do Not Work. One might wish to avoid the inequalities in (New)’s postcondition. Several approaches for this come to mind. First, one could drop the inequalities in (New)’s postcondition, and rely on (Commit)’s postcondition $!(\pi' \text{ contains } \pi)$ to establish (Lock)’s precondition. While this would be sound, in general it is too weak, as we are unable to lock π if we first lock some other object x (because from $!(\pi' \text{ contains } \pi)$ we cannot derive $!(x \cdot \pi' \text{ contains } \pi)$ unless we know $\pi \neq x$). Second, the `Lockset` predicate could be abandoned altogether, using a predicate $\pi.\text{Held}(n)$ instead, that specifies that the current thread holds lock π with reentrancy level n . In particular, $\pi.\text{Held}(0)$ 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 $\ell.\text{Held}(0)$ as the postcondition of (New), replacing the inequalities. However, this approach does not work, because it grants only the object creator permission to lock the created object! While it is possible that a clever program logic could somehow introduce $\pi.\text{Held}(0)$ -predicates in other ways (besides introducing it in the postcondition of (New)), we have not been able to come up with a workable solution along these lines. Besides, it does not solve the aliasing problem.

Wait and notify. Methods `wait`, `notify` and `notifyAll` in class `Object` (introduced in Section 4.2) are specified as follows:

```
class Object{
  pred inv = true;
  requires Lockset(S) * S contains this * inv;
  ensures Lockset(S) * inv;
  final void wait();
  requires Lockset(S) * S contains this;
  ensures Lockset(S);
  final void notify();
  requires Lockset(S) * S contains this;
  ensures Lockset(S);
```

```

final void notifyAll();
}

```

The preconditions for `wait`, `notify` and `notifyAll` require that the receiver is locked, thus ensuring that if a program can be verified, it will never throw an `IllegalMonitorState` exception (or be stuck, according to our formal semantics). The postcondition of `o.wait()` ensures `o.inv`, because `o` is locked again just before `o.wait()` terminates.

Auxiliary Syntax. Recall that in Section 4.2, we added two new head commands `waiting` and `resume` to represent waiting states. The Hoare rules for these commands are as follows:

$$\begin{array}{c}
\frac{\Gamma \vdash \pi, o : \text{lockset}, \text{Object}}{\{\text{Lockset}(\pi) * o.\text{initialized}\}} \quad (\text{Waiting}) \\
\Gamma; r \vdash \quad o.\text{waiting}(n) \\
\{\text{Lockset}(\pi) * o.\text{initialized}\} \\
\\
\frac{\Gamma \vdash o, \pi : \text{Object}, \text{lockset}}{\{\text{Lockset}(\pi) * o.\text{initialized}\}} \quad (\text{Resume}) \\
\Gamma; r \vdash \quad o.\text{resume}(n) \\
\{\text{Lockset}(o^n \cdot \pi) * o.\text{inv}\}
\end{array}$$

In (Resume), o^n denotes the multiset with n occurrences of o . Of course, the rules (Waiting) and (Resume) are never used in source code verification, because source programs do not contain the auxiliary syntax. Instead, the rules (Waiting) and (Resume) are used to state and prove the preservation theorem.

The Thread class. Now we are ready to modify class `Thread` of Section 3's verification system to handle reentrant locks. To handle reentrant locks, we modify class `Thread`'s method contracts as shown in Figure 16. Intuitively, we forbid `fork` and `join`'s contracts (i.e., `preFork` and `postJoin`) to depend on the caller's lockset. This would not make sense since `Lockset` predicates are interpreted w.r.t. to the current thread. Obviously, a thread calling `fork` (or `join`) differs from the newly created (or the joined) thread. We forbid `fork` and `join`'s contracts to depend on the caller's lockset by (1) adding `Lockset(S)` in `fork`'s precondition: because callers of `fork` have to establish `fork`'s precondition, this forbids `preFork` to depend on a `Lockset` predicate (recall that two `Lockset` predicates cannot be *-combined) and (2) by adding `Lockset(S)` in `run`'s postcondition: this forbids `postJoin` to depend on a `Lockset` predicate:

4.5. Verified Programs. We need to update Section 3.4's rules for runtime states to account for reentrant locks. There are two changes to rule (Thread): (1) premise $\text{dom}(\mathcal{H}_{\text{lock}}) = \{o\}$ is added to ensure that a thread's augmented heap only tracks the locks held by this thread and (2) the thread's postcondition is modified to reflect the change in `join`'s postcondition in class `Thread`.

(Thread)

$$\frac{\begin{array}{l} \mathcal{H}_{\text{join}}(o) \leq \llbracket fr \rrbracket \quad \Gamma \vdash \sigma : \Gamma' \quad \Gamma, \Gamma' \vdash s : \diamond \\ \text{cfv}(c) \cap \text{dom}(\Gamma') = \emptyset \quad \text{dom}(\mathcal{H}_{\text{lock}}) = \{o\} \quad \Gamma[\sigma] \vdash \mathcal{E}; \mathcal{H}; s \models F[\sigma] \\ \Gamma, \Gamma'; r \vdash \{F\}c : \text{void}\{ (\text{ex lockset } S) (\text{Lockset}(S)) * fr \cdot o.\text{postJoin}\langle 1 \rangle \} \end{array}}{\mathcal{H} \vdash o \text{ is } (s \text{ in } c) : \diamond}$$

```

class Thread ext Object{
  pred preFork = true;
  group postJoin<perm p> = true;
  requires Lockset(S) * preFork; ensures Lockset(S);
  final void fork();
  requires Join(this,p); ensures postJoin<p>;
  final void join();
  final requires Lockset(nil) * preFork;
    ensures (ex Lockset S)(Lockset(S)) * postJoin<1>;
  void run() { null }
}
    
```

FIGURE 16. Class Thread

We define the set $\text{ready}(\mathcal{H})$ of all initialized objects whose locks are not held, and the function conc that maps abstract lock tables to concrete lock tables:

$$\text{ready}(\mathcal{H}) \triangleq \mathcal{H}_{\text{init}} \setminus \{o \mid (\exists p)(o \in \mathcal{L}(p))\}$$

$$\text{conc}(\mathcal{L})(o) \triangleq \begin{cases} (p, \mathcal{L}(p)(o)) & \text{iff } o \in \mathcal{L}(p) \\ \text{free} & \text{otherwise} \end{cases}$$

In conc 's definition, we let $\mathcal{L}(p)(o)$ stand for the multiplicity of o in $\mathcal{L}(p)$. Note that conc is well-defined, by axiom (f) for augmented heaps (see page 38). The new rule for states ensures that there exists a augmented heap \mathcal{H} to satisfy the thread pool ts and an augmented heap \mathcal{H}' to satisfy the resource invariants of the locks that are ready to be acquired. In addition, function conc relates the program's lock table to the top level augmented heap's abstract lock table:

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

As in Section 3.4, using case analysis on the shape of the $(st : \diamond)$'s proof tree, we have shown that the preservation Theorem 2.4 also holds for the language with reentrant locks. As corollaries we have shown that verified programs satisfy the following properties: null error freeness, partial correctness, and data race freeness (details in [Hur09, Chap. 6]).

Finally, we can also show that an `IllegalMonitorException` cannot occur in a verified program. Suppose we model an `IllegalMonitorException` as a *monitor error* in our language. A head command hc is called a *monitor error* iff it tries to call `wait`, `notify` or `notifyAll` on a lock that is not held. Now we can state the *Monitor Error Free theorem* as a corollary of the preservation theorem for the language with reentrant locks (see page 118 in [Hur09, Chap. 6]).

Theorem 4.1 (Verified Programs are Monitor Error Free). *If $(ct, c) : \diamond$ and $\text{init}(c) \rightarrow_{ct}^* st = \langle h, ts \mid o \text{ is } (s \text{ in } hc; c) \rangle$ then hc is not a monitor error.*

Proof. Similar to Theorem 2.6, the theorem can be proved by contradiction. By $\text{init}(c) : \diamond$ and the preservation theorem, we know that $st : \diamond$. Inspecting the derivation of the cases (Red Wait), (Red Notify), (Red Notify All) (see page 125 of [Hur09, Chapter 6]) shows that it is impossible for a thread to invoke `wait()`, `notify()` or `notifyAll()` if the object is not locked. As these are the only statements that could result in a monitor error, this concludes the proof. □

4.6. Examples of Reasoning with Reentrant Locks. In this section, we show examples of reasoning with reentrant locks. We provide two examples: first we show a specification of a typical container class, using reentrant locks, and the wait-notify mechanism; next we discuss an advanced lock coupling example.

A Typical Container: class Set. For container classes of the Java library, lock reentrancy is crucial to avoid duplication of method implementations, as they typically contain methods that can be called by clients, and by other methods in the container. We illustrate this by discussing a class `Set`, containing a public method `has` that is also called by other methods in the class. Additionally, the container is developed for a concurrent setting, using the wait-notify mechanism when a thread tries to retrieve an object that is not in the container yet.

Class `Set` contains a method `has` that is used to check if some element belongs to the receiver set. In addition, there is a method `add`, which adds an element to the receiver set if not already present. Moreover, it defines a method `visit`, which blocks until a particular element is present in the set. All methods lock the receiver set. Hence, as methods `add` and `visit` call `has`, reentrant locks are crucial for class `Set`'s implementation.

Internally, class `Set` is backed up by a list, shown in Figure 17. Class `List` is a *shallow* container: lists do not have permission to access their values. Instead, values must be accessed by synchronizing on them. That is why the `state` predicate ensures that a list only contains initialized values. Additionally, predicate `state` gives access to field `next` of the list's first node and to all `next` fields of subsequent nodes, and it provides references to the values stored in the list.

Figure 18 presents the implementation of class `Set`, which ensures that an object cannot appear twice in the underlying list. For simplicity, we identify two objects if they have the same address in the heap (i.e., we use Java's `==`)⁵.

The resource invariant of a `Set` consists of (1) the field `rep` and (2) the list pointed to by the field `rep`. This is specified in predicate `inv`'s implementation. A `Set` *owns* its underlying list `rep`: while the receiver set is locked when clients call `has`, `add` or `visit`, the underlying list is never locked. Access rights to the underlying list are packed into the resource invariant of the set (see `inv`'s definition).

Elements of sets should be accessed by synchronizing on them. Although there is no `get` method in class `Set`'s implementation, we make sure that elements of sets are `initialized` (see `state`'s implementation in class `List` and `o.initialized` in various contracts). Hence,

⁵Alternatively, we could put Java's `equals` in class `Object` and use it here.

```

class List extends Object{
  Object element; List next;

  pred state = PointsTo(element,1,v) * PointsTo(next,1,n) *
              v.initialized * n.state;

  requires init * o.initialized; ensures state@List;
  void init(Object o, List n){ element = o; next = n; }

  requires state; ensures state;
  bool has(Object o){
    bool result;
    if(element == o){ result = true; }
    else{ if(next != null){ result = next.has(o); } }
    result;
  }

  requires state * o.initialized; ensures state;
  void add(Object o){ List l = new List; l.init(o,this); }
}

```

FIGURE 17. Class List

a `get` method would have `result.initialized` as a postcondition, allowing clients to lock returned elements.

Method `init` both (1) initializes field `rep` and (2) initializes the set's resource invariant (with the `commit` command). Point (2) is formalized by having `fresh` in `init`'s precondition and having `initialized` in `init`'s postcondition. In addition, `init`'s precondition includes `Set classof this`. This is required to verify that `commit` is sound i.e., that the monitor invariant is established before `commit`.

The contract of method `has` in class `Set` allows lock-reentrant calls. If a lock-reentrant call is performed, however, `inv` is required (as expressed by `(S contains this -* inv)`). Methods `add` and `visit` in class `Set` could be specified similarly allowing lock reentrant calls. Notice that verification of all these methods is straightforward.

A simpler implementation of methods `add` and `visit` in class `Set` would call `has` on the underlying list. In this way, the lock-reentrant call would be avoided. However, our implementation is safer: if method `has` is overridden in subclasses of `Set` (but not method `add` or method `visit`), our implementation is still correct; while the simpler implementation could exhibit unexpected behaviors.

Class `Set` exemplifies a typical use of lock reentrancy and the wait-notify mechanism in the Java library. We believe that our verification system fits well to verify such classes. In addition, this example shows how our system supports programs that include objects that must be locked before access and objects that are accessed without synchronization. Importantly, the addition of locks does not force programmers to indicate `Lockset` predicates everywhere in contracts: class `List`, which backs up class `Set`, does not mention any `Lockset` predicates.

```

class Set extends Object{
  List rep;

  pred inv = PointsTo(rep,1,r) * r.state;

  requires Lockset(S) * init * fresh *
    Set classof this * o.initialized;
  ensures Lockset(S) * !(S contains this) * initialized;
  void init(Object o){ rep = new List; rep.init(o,null); commit; }

  requires Lockset(S) * (S contains this -* inv) * initialized;
  ensures Lockset(S) * (S contains this -* inv);
  bool has(Object o){
    lock(); List result = rep.has(o); unlock(); result;
  }

  requires Lockset(S) * !(S contains this) *
    initialized * o.initialized;
  ensures Lockset(S) * !(S contains this);
  void add(Object o){
    lock(); if(!has(o)){ rep.add(o); notifyAll(); } unlock();
  }

  requires Lockset(S) * !(S contains this) * initialized;
  ensures Lockset(S) * !(S contains this) * initialized;
  void visit(Object o){
    lock(); while(!has(o)){ wait(); } unlock();
  }
}

```

FIGURE 18. Class `Set` in presence of reentrant locks

Finally, suppose class `Set` would be extended by a subclass `BoundedSet` containing a field `count` to keep track of the number of elements stored in the set. The resource invariant of `BoundedSet` would be defined as `pred inv = PointsTo(count,1,-)`, which implicitly would be conjoined with the inherited resource invariant from `Set`. Thus, locking any instance of the `BoundedSet` would provide access to the field `count`, *and* to the underlying list representation of the set.

Lock Coupling. Next we illustrate how our verification system handles lock coupling. We use the following convenient abbreviations:

$$\begin{aligned} \pi.\text{locked}(\pi') &\triangleq \text{Lockset}(\pi \cdot \pi') \\ \pi.\text{unlocked}(\pi') &\triangleq \text{Lockset}(\pi') * !(\pi' \text{ contains } \pi) \end{aligned}$$

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. Each node n maintains the invariant $n.\text{count} == n.\text{next}.\text{count} + 1$.

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

Lock coupling has been used as an example by Gotsman et al. [GBC⁺07] 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:

```
requires this.unlocked(S) * no list node is in S;
ensures Lockset(S);
void insert(int x);
```

The question is how to formally represent the informal condition written 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:

```
requires this.unlocked(S) * no this-owned object is in S;
ensures Lockset(S);
void insert(int x);
```

To express this formally, we define a marker interface, i.e., an interface with no content, for owned objects:

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

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

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

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

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

```
interface SortedIntList {
  pred inv<int c>; // c is the number of list nodes
  requires this.inv<c>; ensures this.inv<c> * result==c;
  int size();

  requires this.unlocked(S) * this.traversable(S);
  ensures Lockset(S);
  void insert(int x);
}
```


Figure 19 shows a tail-recursive lock-coupling implementation of `SortedIntList`. 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'`. Figure 19's implementation has been verified in our system.

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

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

This example demonstrates that we can handle fine-grained concurrency despite the technical difficulties raised by lock reentrancy (i.e., `lock`'s precondition is harder to prove). However, we have to fall back on the type system to verify this example. Consequently, ownership becomes *static*; however based on the design decision of the data structure this is acceptable. Usually when it is necessary that nodes can be transferred from one container to another, all nodes have to come from a dedicated node pool. In that case, our approach would still work, but with the node pool as the owner.

5. RELATED WORK

The work that is closest related to our work is Parkinson's thesis [Par05] (recently represented in [PB13]). This formalizes a subset of singlethreaded Java to specify and verify such programs with separation logic. There are, however, a few differences: we feature value-parameterized classes, we do not include casts (but it would be straightforward to add them, as we did in our earlier work [HH08a]), we do not model constructors, we do not provide block scoping, and, contrary to Parkinson, programs written in our model language are not valid Java programs. While Parkinson introduced abstract predicates and permissions, he does not combine them as we do. Later, both Parkinson and Bierman [PB08] and Chin et al. [CDNQ08] provided a flexible way to handle subclassing.

Separation-logic-based approaches for parallel programs [O'H07, BCO05] focused on a theoretically elegant, but unrealistic, parallel operator. Notable exceptions are Hobor et al. [HAN08] and Gotsman et al. [GBC⁺07] who studied (concurrently to us) Posix threads for C-like programs. Contrary to us, Hobor et al. do not model `join` as a native method, instead they require programmers to model `join` with locks. For verification purposes, this means that Hobor et al. would need extra facilities to make reasoning about `fork/join` as simple as we do. Gotsman et al.'s work is very similar to Hobor et al.'s work.

There are a number of similarities between our work and Gotsman et al. [GBC⁺07]'s work. For instance in 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 cover thread joining in a simpler and more powerful way, because we allow multiple read-only joining. The essential differences between Gotsman's and our paper are (1) that we treat reentrant locks, which are a different

```

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) );
    requires this.inv<c>; ensures this.inv<c> * result==c;
    int size() { return head.count; }
    requires Lockset(S) * !(S contains this) * this.traversable(S);
    ensures 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;
    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 ) );
    public pred inv<int c> = couple<c,c-1>;
    requires PointsTo(next.count, 1/2, c);
    ensures 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; }
    requires Lockset(this.S) * owner.traversable(S) * this.couple<c+1,c-1>;
    ensures 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(); } } }

```

FIGURE 19. A lock-coupling list

synchronization primitive than single-entrant locks, and (2) that we treat subclassing and extension of resource invariants in subclasses. Hobor et al.’s work [HAN08] is very similar to [GBC⁺07].

Zhao in his thesis [Zha07] developed a permission-based type system for a concurrent Java-like language to detect data races and deadlocks. His permission system is an extension of Boyland’s original permission system [Boy03]. Nested permission are used to model protected objects, while guards can be passed as class parameters. The type system handles reentrant locks, but without counting the reentrancy level. Moreover, joins are not supported in his work, and the system can only verify a fixed set of properties, *i.e.*, it has no support for user-specified contracts.

A different approach is taken by Vafeiadis, Parkinson et al. [VP07, WDP10], combining rely/guarantee reasoning with separation logic. On one hand, this is both powerful and flexible: fine-grained concurrent algorithms can be specified and verified. On the other hand, their verification system is more complex than ours. This line of research has been extended by Dodds et al., proposing deny-guarantee reasoning [DFPV09] to tame dynamically scoped threads. The idea of deny-guarantee reasoning is to lift separation logic to assert about the possible interferences between threads. Recently, Concurrent Abstract Predicates (CAP) [DYDG⁺10] have been proposed by Dinsdale-Young et al. as a further follow-up on deny-guarantee reasoning. They proposed a logic by which interferences can be asserted with actions instrumented by permissions. Permission-based actions can describe how a thread can treat the state. Using CAP to specify a mutable data structure, one can distinguish between the *internal* shared states and local states of the data structure. Abstract predicates in CAP encapsulate both resources and interferences which allows one to reason about the client program without having to deal with all the underlying interferences and resources. Initially, it was not possible to use CAP to reason about synchronizer object, because they protect *external* shared resources. However, inspiring from [JP11] and [DJP11] Svendsen extended CAP for higher-order separation logic to specify library usage protocols. The development of CAP and HOCAP seem to be an important progress to reason about concurrent programs. However, there is no well-developed tool support for them yet, the approach does not consider reentrant locks, and it results in a highly-complicated verification technique, especially when it should be applied to a realistic programming language such as Java.

Another related line of work is by Jacobs et al. [JSPS06] who extend the Boogie methodology for reasoning about object invariants [BDF⁺04] 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 concurrent separation logic and our variant of it also do) rather than aiming for a complete program logic. In this approach, objects can be in two states: unshared or shared. Unshared objects can only be accessed by the thread that created them; while shared objects can be accessed by all threads, provided these threads synchronize on this object. This partially correspond to our method: Jacobs et al.’s **shared** objects (objects that are shared between threads) directly correspond to our **initialized** objects (objects whose resource invariants are initialized). While Jacobs et al.’s policy is simple, it is too restrictive: an object cannot be passed by one thread to another thread without requiring the latter thread to synchronize on this object. Jacobs et al.’s system prevents deadlocks, 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.

Smans et al. [SJPS08, SJP09] automatically verify sequential programs using *implicit dynamic frames*. While their approach uses first-order logic, it is close to separation logic, because their verification algorithm approximates the set of locations accessed by methods (like specifications in separation logic). On the upside, Smans et al.’s approach alleviates the burden of specifying the set of locations accessed by methods, because such sets are inferred from functional specifications. Furthermore, (1) like other first-order logic based approaches; they can use off-the-shelf theorem provers and (2) they implemented their approach. On the downside, solving the verification conditions generated by Smans et al.’s tool is much slower than using symbolic execution and separation logic (like [DP08]). Another drawback is that they cannot write specifications that mirrors separation logic’s magic wand $-*$. The magic wand is crucial to specify data structures that temporarily “lend” a part of their representation to clients, like iterators [HH09].

Like Smans et al., Leino and Müller [LM09] presented a verification system for multi-threaded programs that uses implicit dynamic frames and SMT solvers. Contrary to their previous work [JSPS06] they do not impose a programming model: they use fractional permissions to handle concurrency. They do not support multiple readonly joiner threads but they prevent deadlock. Consequently, even if they do not handle reentrant locks, these locks could be handled without a major effort.

Finally, in a more traditional approach, De Boer [dB07] extends the results of Abraham et al. [ÁdBdRS03] with a sound and complete proof system based on the Owicki/Gries method, to generate interference freedom tests for dynamically created threads in Java. In his approach, interferences between threads are annotated as *global* assertions and *local* properties are proved in sequential Hoare logic. Java’s `synchronized` methods are considered as the programs’ synchronization mechanisms and static auxiliary variables are defined to control the owner and reentrancy level of the lock. While this work covers dynamic thread creation, it lacks support for *reentrant locks* and object-oriented features of Java.

6. CONCLUSION

In this paper, we have presented a variant of permission-based separation logic that allows reasoning about object-oriented concurrent programs with dynamic threads and reentrant locks. The main selling point of this logic is that it combines several existing specification techniques, *and* that it is not developed for an idealized programming language. Together this makes it powerful and practical enough to reason about real-life concurrent Java programs, as has been demonstrated on several examples, both in a sequential and in a concurrent setting.

An essential ingredient of the logic is the use of permissions. These ensure that in a verified program, data races cannot occur, while shared readings are allowed. Thus concurrent execution of the program is restricted as little as possible. Further, the logic also contains abstract predicates, as proposed by Parkinson, which are suitable to reason about inheritance, and class parameters. This paper is the first to combine these three different features in a single specification language for a realistic programming language.

Currently, a tool [VCT, ABHZZS12] is being developed for this logic in the context of the VerCors project⁶. Throughout, the tool is developed with practical usability in mind: eventually it should provide sufficient support for a programmer to prove correctness of his or her applications.

To ensure practical usability involves several topics: (1) improving readability of the specification language, for example by merging it with an existing specification language such as JML [LPC⁺07]; (2) development of appropriate proof theories to automatically discharge proof obligations; and (3) development of techniques to reason about the absence of aliasing in the context of lock-reentrancy. The first topic has also been investigated both by Tuerk [Tue09] and Smans et al. [SJP10], while the second topic has been investigated by Parkinson et al. [DP08]. However, in both cases the results have to be further extended to fit in our framework, in particular because they do not consider the magic wand. Concerning the third topic, the lock-coupling example (Section 4.6), uses class parameters to model ownership. We will investigate how this can be done more systematically. At present, (simplified versions of) the examples in this paper can be verified by the tool.

Further we are also extending the application domain of the logic, to be able to reason about a larger class of concurrent Java programs, and to verify also functional properties of these applications. We mention in particular the following recent results and plans for future work:

- We specified the `BlockingQueue` hierarchy from the `java.util.concurrent` library using a history-based specification [ZSHB12]. The specifications can be used to derive functional properties about queues, for example to show that in a concurrent environment the order of elements is always preserved.
- We also developed formal specifications for several synchronization classes, such as the (reentrant and read-write) locks, semaphores and latches from the Java API.
- We are developing techniques to reason about functional properties that have to hold throughout an execution, so-called *strong invariants*.
- We are formally specify classes from `atomic` package from the Java API to support reasoning about lock-free data structures. As a first step we have specified `AtomicInteger` as a primitive synchronizer and proved the correctness of several synchronization patterns on top of this.
- We also plan to investigate whether permission annotations can be generated, instead of being written by the programmer.
- We have been adapting the current logic to reason about GPU kernels [HM13].

ACKNOWLEDGMENTS

We thank Ronald Burgman for working out a first version of the specification of the sequential and parallel mergesort algorithms.

REFERENCES

- [ABD⁺14] Afshin Amighi, Stefan Blom, Saeed Darabi, Marieke Huisman, Wojciech Mostowski, and Marina Zaharieva-Stojanovski. Verification of concurrent systems with vercors. In *SFM*, pages 172–216, 2014.

⁶<http://fmt.cs.utwente.nl/research/projects/VerCors/>

- [ABHZS12] A. Amighi, S. Blom, M. Huisman, and M. Zaharieva-Stojanovski. The vercors project: setting up basecamp. In *Proceedings of the sixth workshop on Programming languages meets program verification*, PLPV '12, pages 71–82, New York, NY, USA, 2012. ACM.
- [ÁdBdRS03] E. Ábrahám, F. S. de Boer, W.-P. de Roever, and M. Steffen. Tool-supported proof system for multithreaded Java. In F. S. de Boer, M. M. Bonsangue, S. Graf, and W.-P. de Roever, editors, *Formal Methods for Components and Objects*, number 2852 in *Lecture Notes in Computer Science*, pages 1–32. Springer-Verlag, 2003.
- [AFF06] M. Abadi, C. Flanagan, and S. Freund. Types for safe locking: Static race detection for Java. *ACM Transactions on Programming Languages and Systems*, 28(2):207–255, 2006.
- [And91] G. Andrews. *Concurrent Programming: Principles and Practice*. Benjamin/Cummings, 1991.
- [BCO05] J. Berdine, C. Calcagno, and P. W. O’Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In F. S. de Boer, M. M. Bonsangue, S. Graf, and W.-P. de Roever, editors, *Formal Methods for Components and Objects*, volume 4111 of *Lecture Notes in Computer Science*, pages 115–137. Springer-Verlag, 2005.
- [BCY05] R. Bornat, C. Calcagno, and H. Yang. Variables as resource in separation logic. In *Mathematical Foundations of Programming Semantics*, volume 155 of *Electronic Notes in Theoretical Computer Science*, pages 247–276. Elsevier, 2005.
- [BDF⁺04] M. Barnett, R. DeLine, M. Fähndrich, K. R. M. Leino, and W. Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3(6):27–56, 2004.
- [BH13] S. Blom and M. Huisman. Witnessing the elimination of magic wands. *STTT (conditionally accepted)*, 2013.
- [BHS07] B. Beckert, R. Hähnle, and P. H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*. Number 4334 in *Lecture Notes in Computer Science*. Springer-Verlag, 2007.
- [BOCP05] R. Bornat, P. W. O’Hearn, C. Calcagno, and M. Parkinson. Permission accounting in separation logic. In J. Palsberg and M. Abadi, editors, *Principles of Programming Languages*, pages 259–270. ACM Press, 2005.
- [Boy03] J. Boyland. Checking interference with fractional permissions. In R. Cousot, editor, *Static Analysis Symposium*, volume 2694 of *Lecture Notes in Computer Science*, pages 55–72. Springer-Verlag, 2003.
- [Boy07] J. Boyland. Semantics of fractional permissions with nesting. Technical report, University of Wisconsin at Milwaukee, December 2007.
- [Bro04] S. Brookes. A semantics for concurrent separation logic. In *Conference on Concurrency Theory*, volume 3170 of *Lecture Notes in Computer Science*, pages 16–34. Springer-Verlag, 2004.
- [CD02] D. G. Clarke and S. Drossopoulou. Ownership, encapsulation and the disjointness of type and effect. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 292–310. ACM Press, 2002.
- [CDNQ08] W. Chin, C. David, H. Nguyen, and S. Qin. Enhancing modular OO verification with separation logic. In G. C. Necula and P. Wadler, editors, *Principles of Programming Languages*, pages 87–99. ACM Press, 2008.
- [CPN98] D. G. Clarke, J. M. Potter, and J. Noble. Ownership types for flexible alias protection. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, volume 33:10 of *ACM SIGPLAN Notices*, pages 48–64, New York, October 1998. ACM Press.
- [CWM99] K. Crary, D. Walker, and G. Morrisett. Typed memory management in a calculus of capabilities. In *Principles of Programming Languages*, pages 262–275, 1999.
- [dB07] 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*, pages 252–268, 2007.
- [DFPV09] M. Dodds, X. Feng, M. Parkinson, and V. Vafeiadis. Deny-guarantee reasoning. In *European Symposium on Programming*, *Lecture Notes in Computer Science*, pages 363–377. Springer-Verlag, 2009.
- [DJP11] Mike Dodds, Suresh Jagannathan, and Matthew J. Parkinson. Modular reasoning for deterministic parallelism. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '11, pages 259–270, New York, NY, USA, 2011. ACM.

- [DP08] D. DiStefano and M. Parkinson. jStar: Towards practical verification for Java. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 213–226. ACM Press, 2008.
- [DYDG⁺10] T. Dinsdale-Young, M. Dodds, P. Gardner, M. Parkinson, and V. Vafeiadis. Concurrent abstract predicates. In *Proc. 24th European Conference on Object-Oriented Programming (ECOOP’10)*, Lecture Notes in Computer Science. Springer, 2010.
- [FLL⁺02] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *Programming Languages Design and Implementation*, volume 37, pages 234–245, June 2002.
- [FQ03] C. Flanagan and S. Qadeer. A type and effect system for atomicity. In *Programming Languages Design and Implementation*, volume 38 of *ACM SIGPLAN Notices*, pages 338–349. ACM Press, May 2003.
- [GBC⁺07] A. Gotsman, J. Berdine, B. Cook, N. Rinetzky, and M. Sagiv. Local reasoning for storable locks and threads. In Z. Shao, editor, *Asian Programming Languages and Systems Symposium*, volume 4807 of *Lecture Notes in Computer Science*, pages 19–37. Springer-Verlag, 2007.
- [GBC11] Alexey Gotsman, Josh Berdine, and Byron Cook. Precision and the conjunction rule in concurrent separation logic. *Electron. Notes Theor. Comput. Sci.*, 276:171–190, September 2011.
- [HAN08] A. Hobor, A. Appel, and F.Z. Nardelli. Oracle semantics for concurrent separation logic. In S. Drossopoulou, editor, *Programming Languages and Systems: Proceedings of the 17th European Symposium on Programming, ESOP 2008*, volume 4960 of *Lecture Notes in Computer Science*, pages 353–367. Springer-Verlag, 2008.
- [HH08a] C. Haack and C. Hurlin. Separation logic contracts for a Java-like language with fork/join. In J. Meseguer and G. Rosu, editors, *Algebraic Methodology and Software Technology*, volume 5140 of *Lecture Notes in Computer Science*, pages 199–215. Springer-Verlag, July 2008.
- [HH08b] C. Haack and C. Hurlin. Separation logic contracts for a Java-like language with fork/join. Technical Report 6430, INRIA, January 2008.
- [HH09] C. Haack and C. Hurlin. Resource usage protocols for iterators. *Journal of Object Technology*, 8(4):55–83, 2009.
- [HHH08] C. Haack, M. Huisman, and C. Hurlin. Reasoning about Java’s reentrant locks. In G. Ramalingam, editor, *Asian Programming Languages and Systems Symposium*, volume 5356 of *Lecture Notes in Computer Science*, pages 171–187. Springer-Verlag, December 2008.
- [HM13] M. Huisman and M. Mihelcic. Specification and verification of gpgpu programs using permission-based separation logic. Technical Report TR-CTIT-13-12, Centre for Telematics and Information Technology, University of Twente, Enschede, March 2013. This work was presented at BYTECODE 2013, March 23, 2013, Rome, Italy.
- [Hoa72] C. A. R. Hoare. Towards a theory of parallel programming. In *Operating Systems Techniques*, pages 61–71, New York, NY, USA, 1972. Academic Press.
- [Hoa74] C. A. R. Hoare. Monitors: an operating system structuring concept. *Communications of the ACM*, 17(10):549–557, 1974.
- [Hui01] M. Huisman. *Reasoning about Java programs in higher order logic using PVS and Isabelle*. PhD thesis, Computing Science Institute, University of Nijmegen, 2001.
- [Hur09] C. Hurlin. *Specification and Verification of Multithreaded Object-Oriented Programs with Separation Logic*. PhD thesis, Université Nice Sophia Antipolis, 2009.
- [IO01] S. Ishtiaq and P. O’Hearn. BI as an assertion language for mutable data structures. In *Principles of Programming Languages*, pages 14–26, 2001.
- [Jon83] C. B. Jones. Tentative steps toward a development method for interfering programs. *ACM Transactions on Programming Languages and Systems*, 5(4):596–619, 1983.
- [JP11] Bart Jacobs and Frank Piessens. Expressive modular fine-grained concurrency specification. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL ’11, pages 271–282, New York, NY, USA, 2011. ACM.
- [JSPS06] B. Jacobs, J. Smans, F. Piessens, and W. Schulte. A statically verifiable programming model for concurrent object-oriented programs. In *International Conference on Formal Engineering Methods*, pages 420–439, 2006.
- [JW06] L. Jia and D. Walker. ILC: A foundation for automated reasoning about pointer programs. In *European Symposium on Programming*, pages 131–145, 2006.

- [LM09] K. R. M. Leino and P. Müller. A basis for verifying multi-threaded programs. In G. Castagna, editor, *European Symposium on Programming*, volume 5502 of *Lecture Notes in Computer Science*, pages 378–393. Springer-Verlag, 2009.
- [LPC⁺07] G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. R. Cok, P. Müller, J. Kiniry, and P. Chalin. *JML Reference Manual*, February 2007. Department of Computer Science, Iowa State University. Available from <http://www.jmlspecs.org>.
- [Mic] Sun Microsystems. Java’s documentation: <http://java.sun.com/>.
- [Mül02] P. Müller. *Modular Specification and Verification of Object-Oriented Programs*, volume 2262 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.
- [NAW06] M. Naik, A. Aiken, and J. Whaley. Effective static race detection for Java. In *Programming Languages Design and Implementation*, pages 308–319. ACM Press, 2006.
- [NPSG09] M. Naik, C. Park, K. Sen, and D. Gay. Effective static deadlock detection. In *ICSE*, pages 386–396, 2009.
- [OG75] S. Owicki and D. Gries. An axiomatic proof technique for parallel programs. *Acta Informatica Journal*, 6:319–340, 1975.
- [O’H07] P. W. O’Hearn. Resources, concurrency and local reasoning. *Theoretical Computer Science*, 375(1–3):271–307, 2007.
- [OP99] P. W. O’Hearn and D. J. Pym. The logic of bunched implications. *Bulletin of Symbolic Logic*, 5(2):215–244, 1999.
- [Par05] M. Parkinson. *Local Reasoning for Java*. PhD thesis, University of Cambridge, 2005.
- [PB05] M. Parkinson and G. Bierman. Separation logic and abstraction. In J. Palsberg and M. Abadi, editors, *Principles of Programming Languages*, pages 247–258. ACM Press, 2005.
- [PB08] M. Parkinson and G. Bierman. Separation logic, abstraction and inheritance. In *Principles of Programming Languages*, pages 75–86. ACM Press, 2008.
- [PB13] M. Parkinson and G. Bierman. *Separation Logic for Object-Oriented Programming*, volume 7850 of *Lecture Notes in Computer Science*. Springer, 2013.
- [Rey02] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Logic in Computer Science*, pages 55–74. IEEE Computer Society, 2002.
- [SJP09] J. Smans, B. Jacobs, and F. Piessens. Implicit dynamic frames: Combining dynamic frames and separation logic. In Sophia Drossopoulou, editor, *European Conference on Object-Oriented Programming*, volume 5653 of *Lecture Notes in Computer Science*, pages 148–172. Springer-Verlag, 2009.
- [SJP10] J. Smans, B. Jacobs, and F. Piessens. Heap-dependent expressions in separation logic. In *Proceedings of the 12th IFIP WG 6.1 international conference and 30th IFIP WG 6.1 international conference on Formal Techniques for Distributed Systems*, FMOODS’10/FORTE’10, pages 170–185, Berlin, Heidelberg, 2010. Springer-Verlag.
- [SJPS08] J. Smans, B. Jacobs, F. Piessens, and W. Schulte. An automatic verifier for Java-like programs based on dynamic frames. In J. L. Fiadeiro and P. Inverardi, editors, *Fundamental Approaches to Software Engineering*, volume 4961 of *Lecture Notes in Computer Science*, pages 261–275. Springer-Verlag, 2008.
- [SWM00] F. Smith, D. Walker, and G. Morrisett. Alias types. In G. Smolka, editor, *European Symposium on Programming*, volume 1782 of *Lecture Notes in Computer Science*, pages 366–381. Springer-Verlag, 2000.
- [Tue09] T. Tuerk. A formalisation of smallfoot in HOL. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Theorem Proving in Higher-Order Logics*, volume 5674 of *Lecture Notes in Computer Science*, pages 469–484. Springer-Verlag, 2009.
- [VCT] Vercors verifier tool. Available from: <http://fmt.ewi.utwente.nl/puptol/vercors-verifier>.
- [VHB⁺03] W. Visser, K. Havelund, G. P. Brat, S. Park, and F. Lerda. Model checking programs. *Automated Software Engineering*, 10(2):203–232, 2003.
- [VP07] V. Vafeiadis and M. J. Parkinson. A marriage of rely/guarantee and separation logic. In L. Caires and V. T. Vasconcelos, editors, *Conference on Concurrency Theory*, volume 4703 of *Lecture Notes in Computer Science*, pages 256–271. Springer-Verlag, 2007.
- [Wad93] P. Wadler. A taste of linear logic. In *Mathematical Foundations of Computer Science*, pages 185–210, 1993.

- [WDP10] J. Wickerson, M. Dodds, and M. Parkinson. Explicit stabilisation for modular rely-guarantee reasoning. In Andrew D. Gordon, editor, *European Symposium on Programming*, volume 6012 of *Lecture Notes in Computer Science*, pages 610–629. Springer-Verlag, 2010.
- [Zha07] Y. Zhao. *Concurrency analysis based on fractional permissions*. PhD thesis, University of Wisconsin at Milwaukee, Milwaukee, WI, USA, 2007.
- [ZSHB12] M. Zaharieva-Stojanovski, M. Huisman, and S. Blom. A history of blockingqueues. In Gordon J. Pace and Anders P. Ravn, editors, *FLACOS*, volume 94 of *EPTCS*, pages 31–35, 2012.

APPENDIX A. AUXILIARY DEFINITIONS

A.1. Definitions of lookup functions.

Field Lookup, $\text{fld}(C\langle\bar{\pi}\rangle) = \bar{T}\bar{f}$:

$$\frac{\text{(Fields Base)} \quad \text{(Fields Ind)} \quad \text{fld}(D\langle\bar{\pi}'[\bar{\pi}/\bar{\alpha}]\rangle) = \bar{T}'\bar{f}' \quad \text{class } C\langle\bar{T}\bar{\alpha}\rangle \text{ ext } D\langle\bar{\pi}'\rangle \text{ impl } \bar{U} \{ \bar{T}\bar{f} \text{ pd}^* \text{ ax}^* \text{ md}^* \}}{\text{fld}(\text{Object}) = \emptyset \quad \text{fld}(C\langle\bar{\pi}\rangle) = (\bar{T}\bar{f})[\bar{\pi}/\bar{\alpha}], \bar{T}'\bar{f}'}$$

Axiom Lookup, $\text{axiom}(t\langle\bar{\pi}\rangle) = F$:

$$\text{axiom}(ax^*) \triangleq \begin{cases} \text{true} & \text{if } ax^* = () \\ F * \text{axiom}(ax^*) & \text{if } ax^* = (\text{axiom } F, ax^*) \end{cases}$$

$$\text{axiom}(\bar{T}) \triangleq \begin{cases} \text{true} & \text{if } \bar{T} = () \text{ or } \bar{T} = (\text{Object}) \\ \text{axiom}(U) * \text{axiom}(\bar{V}) & \text{if } \bar{T} = (U, \bar{V}) \end{cases}$$

$$\frac{\text{(Ax Class)} \quad \text{class } C\langle\bar{T}\bar{\alpha}\rangle \text{ ext } U \text{ impl } \bar{V} \{ \text{fd}^* \text{ pd}^* \text{ ax}^* \text{ md}^* \}}{\text{axiom}(C\langle\bar{\pi}\rangle) = \text{axiom}(ax^*[\bar{\pi}/\bar{\alpha}]) * \text{axiom}((U, \bar{V})[\bar{\pi}/\bar{\alpha}])}$$

$$\frac{\text{(Ax Interface)} \quad \text{interface } I\langle\bar{T}\bar{\alpha}\rangle \text{ ext } \bar{U} \{ \text{pt}^* \text{ ax}^* \text{ mt}^* \}}{\text{axiom}(I\langle\bar{\pi}\rangle) = \text{axiom}(ax^*[\bar{\pi}/\bar{\alpha}]) * \text{axiom}(\bar{U}[\bar{\pi}/\bar{\alpha}])}$$

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.

Method Lookup, $\text{mtype}(m, t\langle\bar{\pi}\rangle) = mt$ and $\text{mbody}(m, C\langle\bar{\pi}\rangle) = (\bar{v}).c$:

$$\frac{\text{(Mlkup Object)} \quad \text{class Object } \{ \dots \langle\bar{T}\bar{\alpha}\rangle \text{ spec } U \text{ m}(\bar{V}\bar{v}) \{c\} \dots \}}{\text{mlkup}(m, \text{Object}) = \langle\bar{T}\bar{\alpha}\rangle \text{ spec } U \text{ m}(\bar{V}\bar{v}) \{c\}}$$

$$\frac{\text{(Mlkup Defn)} \quad \text{class } C\langle\bar{T}'\bar{\alpha}'\rangle \text{ ext } U' \text{ impl } \bar{V}' \{ \dots \langle\bar{T}\bar{\alpha}\rangle \text{ spec } U \text{ m}(\bar{V}\bar{v}) \{c\} \dots \}}{\text{mlkup}(m, C\langle\bar{\pi}\rangle) = (\langle\bar{T}\bar{\alpha}\rangle \text{ spec } U \text{ m}(\bar{V}\bar{v}) \{c\})[\bar{\pi}/\bar{\alpha}]}$$

$$\frac{\text{(Mlkup Inherit)} \quad m \notin \text{dom}(md^*) \quad \text{class } C\langle\bar{T}\bar{\alpha}\rangle \text{ ext } D\langle\bar{\pi}'\rangle \text{ impl } \bar{U} \{ \text{fd}^* \text{ pd}^* \text{ md}^* \}}{\text{mlkup}(m, D\langle\bar{\pi}'[\bar{\pi}/\bar{\alpha}]\rangle) = md'}$$

$$\text{mlkup}(m, C\langle\bar{\pi}\rangle) = md'$$

If $\text{mlkup}(m, C\langle\bar{\pi}\rangle) = \langle\bar{T}\bar{\alpha}\rangle \text{requires } F; \text{ensures } G; U\ m(\bar{V}\bar{v})\{c\}$, then:

$$\begin{aligned} \text{mbody}(m, C\langle\bar{\pi}\rangle) &\triangleq (\text{this}; \bar{v}).c \\ \text{mtype}(m, C\langle\bar{\pi}\rangle) &\triangleq \langle\bar{T}\bar{\alpha}\rangle \text{requires } F; \text{ensures } (\text{result}); U\ m(C\langle\bar{\pi}\rangle\ \text{this}; \bar{V}\bar{v}) \end{aligned}$$

(Mtype Interface)

$$\frac{\text{interface } I\langle\bar{T}\bar{\alpha}\rangle \text{ ext } \bar{U}\ \{\dots\ \langle\bar{T}'\bar{\alpha}'\rangle \text{requires } F; \text{ensures } G; U'\ m(\bar{V}'\bar{v}'); \dots\}}{\text{mtype}(m, I\langle\bar{\pi}\rangle) = (\langle\bar{T}'\bar{\alpha}'\rangle \text{requires } F; \text{ensures } (\text{result}); U'\ m(I\langle\bar{\pi}\rangle\ \text{this}; \bar{V}'\bar{v}'))[\bar{\pi}/\bar{\alpha}]}$$

(Mtype Interface Inherit)

$$\frac{\text{interface } I\langle\bar{T}\bar{\alpha}\rangle \text{ ext } \bar{U}, V, \bar{U}'\ \{pt^*\ ax^*\ mt^*\} \\ m \notin \text{dom}(mt^*) \quad (\forall U \in \bar{U}, \bar{U}')(\text{mtype}(m, U[\bar{\pi}/\bar{\alpha}]) = \text{undef}) \quad \text{mtype}(m, V[\bar{\pi}/\bar{\alpha}]) = mt}{\text{mtype}(m, I\langle\bar{\pi}\rangle) = mt}$$

(Mtype Interface Inherit Object)

$$\frac{\text{interface } I\langle\bar{T}\bar{\alpha}\rangle \text{ ext } \bar{U}\ \{pt^*\ ax^*\ mt^*\} \\ m \notin \text{dom}(mt^*) \quad (\forall U \in \bar{U})(\text{mtype}(m, U[\bar{\pi}/\bar{\alpha}]) = \text{undef}) \quad \text{mtype}(m, \text{Object}) = mt}{\text{mtype}(m, I\langle\bar{\pi}\rangle) = mt}$$

Remarks on predicate lookup:

- The “ext Object” in $\text{plkup}(\text{init}, \text{Object})$ and (Plkup Object) 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`), $\text{df}(T)$ is the default value of type T (df is formally defined in Section 2.2).

Predicate Lookup, $\text{ptype}(P, t\langle\bar{\pi}\rangle) = pt$ and $\text{pbody}(\pi.P\langle\bar{\pi}'\rangle, C\langle\bar{\pi}''\rangle) = F \text{ ext } T$:

$$\text{plkup}(\text{init}, \text{Object}) = \text{pred init} = \text{true ext Object}$$

(Plkup Object)

$$\frac{\text{class Object } \{\dots\ \text{pred } P\langle\bar{T}\bar{\alpha}\rangle = F; \dots\}}{\text{plkup}(P, \text{Object}) = \text{pred } P\langle\bar{T}\bar{\alpha}\rangle = F \text{ ext Object}}$$

(Plkup Defn)

$$\frac{\text{class } C\langle\bar{T}'\bar{\alpha}'\rangle \text{ ext } U \text{ impl } \bar{V}\ \{\dots\ \text{pred } P\langle\bar{T}\bar{\alpha}\rangle = F; \dots\}}{\text{plkup}(P, C\langle\bar{\pi}\rangle) = (\text{pred } P\langle\bar{T}\bar{\alpha}\rangle = F \text{ ext Object})[\bar{\pi}/\bar{\alpha}']}$$

(Plkup `init`)

$$\frac{\text{class } C\langle\bar{T}'\bar{\alpha}'\rangle \text{ ext } U \text{ impl } \bar{V}\ \{fd^*\ pd^*\ md^*\} \quad F = \otimes_{T\ f \in fd} \text{*PointsTo}(\text{this}.f, 1, \text{df}(T))}{\text{plkup}(\text{init}, C\langle\bar{\pi}\rangle) = (\text{pred init} = F \text{ ext } U)[\bar{\pi}/\bar{\alpha}]}$$

(Plkup Inherit) $P \notin \text{dom}(pd^*)$

$$\frac{\text{class } C\langle\bar{T}'\bar{\alpha}'\rangle \text{ ext } U \text{ impl } \bar{V}\ \{fd^*\ pd^*\ md^*\} \quad \text{plkup}(P, U) = \text{pred } P\langle\bar{T}\bar{\alpha}\rangle = F \text{ ext } U'}{\text{plkup}(P, C\langle\bar{\pi}\rangle) = (\text{pred } P\langle\bar{T}\bar{\alpha}\rangle = \text{true ext } U)[\bar{\pi}/\bar{\alpha}]}$$

If $\text{plkup}(P, C\langle\bar{\pi}\rangle) = \text{pred } P\langle\bar{T}\bar{\alpha}\rangle = F \text{ ext } V$, then:

$$\begin{aligned} \text{pbody}(\pi.P\langle\bar{\pi}'\rangle, C\langle\bar{\pi}\rangle) &\triangleq (F \text{ ext } V)[\pi/\text{this}, \bar{\pi}'/\bar{\alpha}] \\ \text{ptype}(P, C\langle\bar{\pi}\rangle) &\triangleq \text{pred } P\langle\bar{T}\bar{\alpha}\rangle \end{aligned}$$

(Ptype Interface)

$$\frac{\text{interface } I\langle\bar{T}\bar{\alpha}\rangle \text{ ext } \bar{U}\ \{\dots\ \text{pred } P\langle\bar{T}'\bar{\alpha}'\rangle; \dots\}}{\text{ptype}(P, I\langle\bar{\pi}\rangle) = (\text{pred } P\langle\bar{T}'\bar{\alpha}'\rangle)[\bar{\pi}/\bar{\alpha}]}$$

$$\begin{array}{c}
\text{(Ptype Interface Inherit)} \quad \text{interface } I\langle\bar{T} \bar{\alpha}\rangle \text{ ext } \bar{U}, V, \bar{U}' \{pt^* \ ax^* \ mt^*\} \\
\hline
P \notin \text{dom}(S) \quad (\forall U \in \bar{U}, \bar{U}') (\text{ptype}(P, U[\bar{\pi}/\bar{\alpha}]) = \text{undef}) \quad \text{ptype}(P, V[\bar{\pi}/\bar{\alpha}]) = pt \\
\hline
\text{ptype}(P, I\langle\bar{\pi}\rangle) = pt \\
\\
\text{(Ptype Interface Inherit Object)} \quad \text{interface } I\langle\bar{T} \bar{\alpha}\rangle \text{ ext } \bar{U} \{pt^* \ ax^* \ mt^*\} \\
\hline
P \notin \text{dom}(pt^*) \quad (\forall U \in \bar{U}) (\text{ptype}(P, U[\bar{\pi}/\bar{\alpha}]) = \text{undef}) \quad \text{ptype}(P, \text{Object}) = pt \\
\hline
\text{ptype}(P, I\langle\bar{\pi}\rangle) = pt
\end{array}$$

The partial function $\text{ptype}(P, t\langle\bar{\pi}\rangle)$ is extended to predicate selectors $P@C$ as follows:

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

A.2. Semantics of operators. To define the semantics of the command assigning the result of an operation (case $\ell = op(\bar{v})$ of our command language), we define the semantics of operators.

Let arity be a function that assigns to each operator its arity. We define:

$$\begin{array}{l}
\text{arity}(==) \triangleq 2 \quad \text{arity}(\&) \triangleq 2 \quad \text{arity}(!) \triangleq 2 \\
\text{arity}(!) \triangleq 1 \quad \text{arity}(C \text{ classof}) \triangleq 1 \quad \text{arity}(\text{instanceof } T) \triangleq 1
\end{array}$$

Let type be a function that maps each operator op to a partial function $\text{type}(op)$ of type $\{\text{int}, \text{bool}, \text{Object}, \text{perm}\}^{\text{arity}(op)} \rightarrow \{\text{int}, \text{bool}, \text{perm}\}$. We define:

$$\begin{array}{l}
\text{type}(==) \triangleq \{ ((T, T), \text{bool}) \mid T \in \{\text{int}, \text{bool}, \text{Object}, \text{perm}, \text{lockset}\} \} \\
\text{type}(!) \triangleq \{ (\text{bool}, \text{bool}) \} \quad \text{type}(\&) \triangleq \text{type}(!) \triangleq \{ ((\text{bool}, \text{bool}), \text{bool}) \} \\
\text{type}(C \text{ classof}) \triangleq \{ (\text{Object}, \text{bool}) \} \\
\text{type}(\text{instanceof } T) \triangleq \{ (\text{Object}, \text{bool}) \}
\end{array}$$

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

$$\llbracket op \rrbracket \in \text{Heap} \rightarrow \bigcup_{(\bar{T}, U) \in \text{type}(op)} \llbracket \bar{T} \rrbracket \rightarrow \llbracket U \rrbracket$$

For the logical operators $!$, $|$ and $\&$, we assume the usual interpretations. Operator $==$ is interpreted as the identity relation. The semantics of isclassof and instanceof is as follows:

$$\begin{array}{l}
\llbracket C \text{ classof} \rrbracket^h(o) \triangleq \begin{cases} \text{true} & \text{if } o \neq \text{null} \text{ and } h(o)_1 = C\langle\bar{\pi}\rangle \text{ for some } \bar{\pi} \\ \text{false} & \text{if } o \neq \text{null}, h(o)_1 = D\langle\bar{\pi}\rangle, \text{ and } D \neq C \\ \text{false} & \text{if } o = \text{null} \\ \text{undef} & o \notin \text{dom}(h) \end{cases} \\
\llbracket o \text{ instanceof } T \rrbracket^h \triangleq \begin{cases} \text{true} & \text{if } o \neq \text{null} \text{ and } h(o)_1 <: T \\ \text{false} & \text{if } o \neq \text{null} \text{ and } h(o)_1 \not<: T \\ \text{false} & \text{if } o = \text{null} \\ \text{undef} & \text{if } o \notin \text{dom}(h) \end{cases}
\end{array}$$

Formally, the semantics of operators is expressed as follows:

Semantics of Operators: $\llbracket op(\bar{v}) \rrbracket : \text{Heap} \rightarrow \text{Stack} \rightarrow \text{Val}$:

| |
|--|
| (Sem Op) $\frac{\llbracket w_1 \rrbracket_s^h = v_1 \quad \cdots \quad \llbracket w_n \rrbracket_s^h = v_n \quad \llbracket op \rrbracket^h(v_1, \dots, v_n) = v}{\llbracket op(w_1, \dots, w_n) \rrbracket_s^h = v}$ |
|--|

A.3. Semantics of specification values and expressions. Expressions contain specification values, read-write variables, and operators. Therefore, we give the semantics of specification values and the semantics of expressions together. Let **SemVal** be the semantic domain of specification values. For the moment, **SemVal** is simply **Val**; but it is extended in Sec. 2.3.2 as we extend specification values. We range over **SemVal** with meta-variable μ .

Semantics of Specification Values and Expressions, $\llbracket e \rrbracket : \text{Heap} \rightarrow \text{Stack} \rightarrow \text{SemVal}$:

| | | |
|---|--|--|
| (Sem SpecVal) | (Sem Var) | (Sem Op) |
| $\frac{\llbracket \pi \rrbracket = \mu}{\llbracket \pi \rrbracket_s^h = \mu}$ | $\frac{s(\ell) = v}{\llbracket \ell \rrbracket_s^h = v}$ | $\frac{\llbracket w_1 \rrbracket_s^h = v_1 \quad \cdots \quad \llbracket w_n \rrbracket_s^h = v_n \quad \llbracket op \rrbracket^h(v_1, \dots, v_n) = v}{\llbracket op(w_1, \dots, w_n) \rrbracket_s^h = v}$ |

Note that, we do not have to define a semantics of logical variables α , because we deal with them by substitution.

A.4. Small-step reduction. The state reduction relation \rightarrow_{ct} is given with respect to a class table ct in Section 2.2. 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) | $l \notin \text{dom}(s) \quad s' = s[l \mapsto \text{df}(T)]$ $\langle h, p \text{ is } (s \text{ in } T \ell; c) \rangle \rightarrow \langle h, p \text{ is } (s' \text{ in } c) \rangle$ |
| (Red Fin Dcl) | $s(\ell) = v \quad c' = c[v/i]$ $\langle h, p \text{ is } (s \text{ in } T \ell = v; c) \rangle \rightarrow \langle h, p \text{ is } (s \text{ in } c') \rangle$ |
| (Red Var Set) | $s' = s[\ell \mapsto v]$ $\langle h, p \text{ is } (s \text{ in } \ell = v; c) \rangle \rightarrow \langle h, p \text{ is } (s' \text{ in } c) \rangle$ |
| (Red Op) | $\text{arity}(op) = \bar{v} \quad \llbracket op \rrbracket^h(\bar{v}) = w \quad s' = s[\ell \mapsto w]$ $\langle h, p \text{ is } (s \text{ in } \ell = op(\bar{v}); c) \rangle \rightarrow \langle h, p \text{ is } (s' \text{ in } c) \rangle$ |
| (Red Get) | $s' = s[\ell \mapsto h(o)_2(f)]$ $\langle h, p \text{ is } (s \text{ in } \ell = o.f; c) \rangle \rightarrow \langle h, p \text{ is } (s' \text{ in } c) \rangle$ |
| (Red Set) | $h' = h[o.f \mapsto v]$ $\langle h, p \text{ is } (s \text{ in } o.f = v; c) \rangle \rightarrow \langle h', p \text{ is } (s \text{ in } c) \rangle$ |
| (Red New) | $o \notin \text{dom}(h) \quad h' = h[o \mapsto (C\langle \bar{\pi} \rangle, \text{initStore}(C\langle \bar{\pi} \rangle))]$ $s' = s[\ell \mapsto o] \quad l' = l[o \mapsto \text{free}]$ $\langle h, l, ts \mid p \text{ is } (s \text{ in } \ell = \text{new } C\langle \bar{\pi} \rangle; c) \rangle \rightarrow \langle h', l', ts \mid p \text{ is } (s' \text{ in } c) \rangle$ |
| (Red Call) | $h(o)_1 = C\langle \bar{\pi} \rangle \quad \text{mbody}(m, C\langle \bar{\pi} \rangle) = (i_0; \bar{v}).c_m \quad c' = c_m[o/i_0, \bar{v}/\bar{v}]$ $\langle h, p \text{ is } (s \text{ in } \ell = o.m(\bar{v}); c) \rangle \rightarrow \langle h, p \text{ is } (s \text{ in } \ell \leftarrow c'; c) \rangle$ |
| (Red Return) | $\langle h, p \text{ is } (s \text{ in } \ell = \text{return}(v); c) \rangle \rightarrow \langle h, p \text{ is } (s \text{ in } \ell = v; c) \rangle$ |

| | |
|-------------------|---|
| (Red If True) | $\langle h, p \text{ is } (s \text{ in if } (\text{true})\{c\}\text{else}\{c'\}; c'') \rangle \rightarrow \langle h, p \text{ is } (s \text{ in } c; c'') \rangle$ |
| (Red If False) | $\langle h, p \text{ is } (s \text{ in if } (\text{false})\{c\}\text{else}\{c'\}; c'') \rangle \rightarrow \langle h, p \text{ is } (s \text{ in } c'; c'') \rangle$ |
| (Red While True) | $\llbracket e \rrbracket_s^h = \text{true}$ $\langle h, p \text{ is } (s \text{ in while } (e)\{c\}; c'') \rangle \rightarrow \langle h, p \text{ is } (s \text{ in } c; \text{while } (e)\{c\}; c'') \rangle$ |
| (Red While False) | $\llbracket e \rrbracket_s^h = \text{false}$ $\langle h, p \text{ is } (s \text{ in while } (e)\{c\}; c'') \rangle \rightarrow \langle h, p \text{ is } (s \text{ in } c') \rangle$ |

A.5. Typing rules. Here we define typing rules needed for Section 2.2, Section 3 and Section 4.

Rules for Section 2.2. Because the semantics of formulas depends on a typing judgment, we need to define typing rules before giving the formulas' semantics.

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

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

A type environment is *good* when objects within its domain are well-typed:

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

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

We define a sanity condition on types: primitive types are always sane, while user-defined types must be such that (1) type identifiers are in the class table and (2) type parameters are well-typed. Below, the existential quantification in (Ty Ref)'s second premise enforces typing derivations to be finite.

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

| | |
|---|---|
| (Ty Primitive) | (Ty Ref) |
| $T \in \{\text{void}, \text{int}, \text{bool}, \text{perm}\}$ | $t < \bar{T} \bar{\alpha} > \in ct$ $(\exists \Gamma' \subset \Gamma)(\Gamma' \vdash \diamond \quad \Gamma' \vdash \bar{\pi} : \bar{T}[\bar{\pi}/\bar{\alpha}])$ |
| $\Gamma \vdash T : \diamond$ | $\Gamma \vdash t < \bar{\pi} > : \diamond$ |

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

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

As models of formulas are tuples that contain a heap and a stack (see Section 2.3.1), we define a well-typedness judgment for objects, heaps, and stacks:

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

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

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

Below, we use function $\text{fst} : \text{Heap} \rightarrow (\text{ObjId} \rightarrow \text{Type})$ to extract the function that maps object identifiers to their dynamic types from a heap:

$$h(o) = (T, _) \Rightarrow \text{fst}(h)(o) = T$$

We now define well-typed heaps and stacks:

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

(Heap)

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

(Stack)

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

Because formulas include expressions, we define a well-typedness judgment for values, specification values, and expressions (recall that expressions include specification values of type `bool`).

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

| | | | |
|--|--|--------------------------------------|--|
| (Val Var) | (Val Oid) | (Val Sub) | (Val Null) |
| $\Gamma \vdash \diamond \quad \Gamma(x) = T$ | $\Gamma \vdash \diamond \quad \Gamma(o) = T$ | $\Gamma \vdash \pi : T \quad T <: U$ | $\Gamma \vdash t\langle\bar{\pi}\rangle : \diamond$ |
| $\Gamma \vdash x : T$ | $\Gamma \vdash o : T$ | $\Gamma \vdash \pi : U$ | $\Gamma \vdash \text{null} : t\langle\bar{\pi}\rangle$ |
| (Val Int) | (Val Bool) | (Val Full) | (Val Split) |
| $\Gamma \vdash \diamond$ | $\Gamma \vdash \diamond$ | $\Gamma \vdash \diamond$ | $\Gamma \vdash \pi : \text{perm}$ |
| $\Gamma \vdash n : \text{int}$ | $\Gamma \vdash b : \text{bool}$ | $\Gamma \vdash 1 : \text{perm}$ | $\Gamma \vdash \text{split}(\pi) : \text{perm}$ |

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

| | | |
|------------------------------------|---|--|
| (Exp Sub) | (Exp Var) | (Exp Op) |
| $\Gamma \vdash e : T \quad T <: U$ | $\Gamma \vdash \diamond \quad \Gamma(\ell) = T$ | $\Gamma \vdash \bar{e} : \bar{U} \quad \text{type}(op)(\bar{U}) = T$ |
| $\Gamma \vdash e : U$ | $\Gamma \vdash \ell : T$ | $\Gamma \vdash op(\bar{e}) : T$ |

We now have all the machinery to define well-typed formulas. Below, the partial function $\text{ptype}(P, C\langle\bar{\pi}\rangle)$ (formally defined in A) looks up the type of predicate P in the least supertype of $C\langle\bar{\pi}\rangle$ that defines or extends P .

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

| | |
|--|---|
| (Form Bool) | (Form Points To) |
| $\Gamma \vdash e : \text{bool}$ | $\Gamma \vdash e : U \quad \Gamma \vdash \pi : \text{perm} \quad T f \in \text{fld}(U) \quad \Gamma \vdash e' : T$ |
| $\Gamma \vdash e : \diamond$ | $\Gamma \vdash \text{PointsTo}(e.f, \pi, e') : \diamond$ |
| (Form Log Op) | (Form Pred) |
| $\Gamma \vdash F, F' : \diamond$ | $\Gamma \vdash \pi : U \quad \text{ptype}(\kappa, U) = \text{pred } P\langle\bar{T} \bar{\alpha}\rangle \quad \Gamma \vdash \bar{\pi}' : \bar{T}$ |
| $\Gamma \vdash F \text{ lop } F' : \diamond$ | $\Gamma \vdash \pi.\kappa\langle\bar{\pi}'\rangle : \diamond$ |

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

Rules for Section 3. To cover Section 3's Join formula, we extend the judgment for well-typed formulas as follows:

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

$$\frac{\text{(Form Join)} \quad \Gamma \vdash e : \text{Thread} \quad \Gamma \vdash \pi : \text{perm}}{\Gamma \vdash \text{Join}(e, \pi) : \diamond} \quad \dots$$

Rules for Section 4. To accommodate Section 4.3's lockset's type, we update the previous typing rule for good types:

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

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

The following typing rule extends typing to values representing locksets:

$$\frac{\mu \in \text{Bag}(\text{ObjId})}{\Gamma \vdash \mu : \text{lockset}}$$

To cover formulas about locksets and the state of locks, we extend the judgment for well-typed formulas:

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

$$\frac{\text{(Form Lockset)} \quad \Gamma \vdash \pi : \text{lockset}}{\Gamma \vdash \text{Lockset}(\pi) : \diamond} \quad \frac{\text{(Form Contains)} \quad \Gamma \vdash \pi, e : \text{lockset}, \text{Object}}{\Gamma \vdash \pi \text{ contains } e : \diamond} \quad \frac{\text{(Form Fresh)} \quad \Gamma \vdash e : \text{Object}}{\Gamma \vdash e.\text{fresh} : \diamond}$$

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

A.6. Verification. In this section we present a complete list of natural deduction rules (Section 2.4.1) and Hoare rules (Section 2.4.2). In addition, the formal definitions of the sanity conditions required for the verification of the interfaces and classes (see Section 2.5) will be given.

Logical Consequences.

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

| | | |
|---|--|---|
| $\frac{(\text{Id}) \quad \Gamma \vdash v, \bar{F}, G : \text{Object}, \diamond}{\Gamma; v; \bar{F}, G \vdash G}$ | $\frac{(\text{Ax}) \quad \Gamma \vdash v, \bar{F}, G : \text{Object}, \diamond}{\Gamma; v; \bar{F} \vdash G}$ | |
| $\frac{(\ast \text{ Intro}) \quad \Gamma; v; \bar{F} \vdash H_1 \quad \Gamma; v; \bar{G} \vdash H_2}{\Gamma; v; \bar{F}, \bar{G} \vdash H_1 \ast H_2}$ | $\frac{(\ast \text{ Elim}) \quad \Gamma; v; \bar{F} \vdash G_1 \ast G_2 \quad \Gamma; v; \bar{E}, G_1, G_2 \vdash H}{\Gamma; v; \bar{F}, \bar{E} \vdash H}$ | |
| $\frac{(-\ast \text{ Intro}) \quad \Gamma; v; \bar{F}, G_1 \vdash G_2}{\Gamma; v; \bar{F} \vdash G_1 -\ast G_2}$ | $\frac{(-\ast \text{ Elim}) \quad \Gamma; v; \bar{F} \vdash H_1 -\ast H_2 \quad \Gamma; v; \bar{G} \vdash H_1}{\Gamma; v; \bar{F}, \bar{G} \vdash H_2}$ | |
| $\frac{(\& \text{ Intro}) \quad \Gamma; v; \bar{F} \vdash G_1 \quad \Gamma; v; \bar{F} \vdash G_2}{\Gamma; v; \bar{F} \vdash G_1 \& G_2}$ | $\frac{(\& \text{ Elim 1}) \quad \Gamma; v; \bar{F} \vdash G_1 \& G_2}{\Gamma; v; \bar{F} \vdash G_1}$ | $\frac{(\& \text{ Elim 2}) \quad \Gamma; v; \bar{F} \vdash G_1 \& G_2}{\Gamma; v; \bar{F} \vdash G_2}$ |
| $\frac{(\text{ Intro 1}) \quad \Gamma; v; \bar{F} \vdash G_1}{\Gamma; v; \bar{F} \vdash G_1 G_2}$ | $\frac{(\text{ Intro 2}) \quad \Gamma; v; \bar{F} \vdash G_2}{\Gamma; v; \bar{F} \vdash G_1 G_2}$ | $\frac{(\text{ Elim}) \quad \Gamma; v; \bar{F} \vdash G_1 G_2 \quad \Gamma; v; \bar{E}, G_1 \vdash H \quad \Gamma; v; \bar{E}, G_2 \vdash H}{\Gamma; v; \bar{F}, \bar{E} \vdash H}$ |
| $\frac{(\text{Ex Intro}) \quad \Gamma, \alpha : T \vdash G : \diamond \quad \Gamma \vdash \pi : T \quad \Gamma; v; \bar{F} \vdash G[\pi/\alpha]}{\Gamma; v; \bar{F} \vdash (\text{ex } T \alpha)(G)}$ | $\frac{(\text{Ex Elim}) \quad \alpha \notin \bar{F}, H \quad \Gamma; v; \bar{E} \vdash (\text{ex } T \alpha)(G) \quad \Gamma, \alpha : T; v; \bar{F}, G \vdash H}{\Gamma; v; \bar{E}, \bar{F} \vdash H}$ | |
| $\frac{(\text{Fa Intro}) \quad \alpha \notin \bar{F} \quad \Gamma, \alpha : T; v; \bar{F} \vdash G}{\Gamma; v; \bar{F} \vdash (\text{fa } T \alpha)(G)}$ | $\frac{(\text{Fa Elim}) \quad \Gamma; v; \bar{F} \vdash (\text{fa } T \alpha)(G) \quad \Gamma \vdash \pi : T}{\Gamma; v; \bar{F} \vdash G[\pi/\alpha]}$ | |

Hoare Rules.

Hoare Rules

| | |
|--|-----------|
| $\frac{\Gamma \vdash u, w : U, W \quad W f \in \text{fld}(U)}{\Gamma; v \vdash \{\text{PointsTo}(u, f, 1, W)\} u.f = w \{\text{PointsTo}(u, f, 1, w)\}}$ | (Fld Set) |
| $\frac{\Gamma \vdash u, \pi, w : U, \text{perm}, W \quad W f \in \text{fld}(U) \quad W <: \Gamma(\ell)}{\Gamma; v \vdash \{\text{PointsTo}(u, f, \pi, w)\} \ell = u.f \{\text{PointsTo}(u, f, \pi, w) \ast \ell == w\}}$ | (Get) |
| $\frac{C \langle \bar{T} \bar{\alpha} \rangle \in ct \quad \Gamma \vdash \bar{\pi} : \bar{T}[\bar{\pi}/\bar{\alpha}] \quad C \langle \bar{\pi} \rangle <: \Gamma(\ell)}{\Gamma; v \vdash \{\text{true}\} \ell = \text{new } C \langle \bar{\pi} \rangle \{\ell.\text{init} \ast C \text{ classof } \ell\}}$ | (New) |
| $\text{mtype}(m, t \langle \bar{\pi} \rangle) = \langle \bar{T} \bar{\alpha} \rangle \text{ requires } G; \text{ ensures } (\alpha')(G');$ $U m (t \langle \bar{\pi} \rangle w, \bar{W} \bar{i})$ | |
| $\frac{\sigma = (u/\iota_0, \bar{\pi}'/\bar{\alpha}, \bar{w}/\bar{i}) \quad \Gamma \vdash u, \bar{\pi}', \bar{w} : t \langle \bar{\pi} \rangle, \bar{T}[\sigma], \bar{W}[\sigma] \quad U[\sigma] <: \Gamma(\ell)}{\Gamma; v \vdash \{u != \text{null} \ast G[\sigma]\} \ell = u.m(\bar{w}) \{\text{ex } U[\sigma] \alpha' \} (\alpha' == \ell \ast G'[\sigma])}$ | (Call) |
| $\frac{\Gamma; v; F \vdash G[w/\alpha] \quad \Gamma \vdash w : U <: T \quad \Gamma, \alpha : U \vdash G : \diamond}{\Gamma; v \vdash \{F\} w : T\{U \alpha\}(G)}$ | (Val) |

$$\begin{array}{c}
\frac{\ell \notin F, G \quad \Gamma, \ell : T; v \vdash \{F * \ell == \text{df}(T)\}c : U\{G\}}{\Gamma; v \vdash \{F\}T \ell; c : U\{G\}} \quad (\text{Dcl}) \\
\\
\frac{\iota \notin F, G, v \quad \Gamma \vdash \ell : T \quad \Gamma, \iota : T; v \vdash \{F * \iota == \ell\}c : U\{G\}}{\Gamma; v \vdash \{F\}T \iota = \ell; c : U\{G\}} \quad (\text{Fin Dcl}) \\
\\
\frac{\Gamma; v \vdash \{F\}hc\{F'\} \quad \Gamma; v \vdash \{F'\}c : T\{G\}}{\Gamma; v \vdash \{F\}hc; c : T\{G\}} \quad (\text{Seq}) \\
\\
\frac{\Gamma; v \vdash \{F\}hc\{G\} \quad \Gamma \vdash H : \diamond \quad \text{fv}(H) \cap \text{writes}(hc) = \emptyset}{\Gamma; v \vdash \{F * H\}hc\{G * H\}} \quad (\text{Frame}) \\
\\
\frac{\Gamma; v \vdash \{F'\}hc\{G'\} \quad \Gamma; v; F \vdash F' \quad \Gamma; v; G' \vdash G}{\Gamma; v \vdash \{F\}hc\{G\}} \quad (\text{Consequence}) \\
\\
\frac{\Gamma, \alpha : T; v \vdash \{F\}hc\{G\}}{\Gamma; v \vdash \{(\text{ex } T \alpha)(F)\}hc\{(\text{ex } T \alpha)(G)\}} \quad (\text{Exists}) \\
\\
\frac{\Gamma \vdash w : \Gamma(\ell)}{\Gamma; v \vdash \{\text{true}\}\ell = w\{\ell == w\}} \quad (\text{Var Set}) \\
\\
\frac{\Gamma \vdash \text{op}(\bar{w}) : \Gamma(\ell)}{\Gamma; v \vdash \{\text{true}\}\ell = \text{op}(\bar{w})\{\ell == \text{op}(\bar{w})\}} \quad (\text{Op}) \\
\\
\frac{\Gamma \vdash w : \text{bool} \quad \Gamma; v \vdash \{F * w\}c : \text{void}\{G\} \quad \Gamma; v \vdash \{F * !w\}c' : \text{void}\{G\}}{\Gamma; v \vdash \{F\}\text{if}(w)\{c\}\text{else}\{c'\}\{G\}} \quad (\text{If}) \\
\\
\frac{\Gamma \vdash e, F : \text{bool}, \diamond \quad \Gamma; v \vdash \{F \& e\}c : \text{void}\{F\}}{\Gamma; v \vdash \{F\}\text{invariant } F; \text{while}(e)\{c\} : \text{void}\{F \& !e\}} \quad (\text{While}) \\
\\
\frac{\Gamma; v; F \vdash G}{\Gamma; v \vdash \{F\}\text{assert}(G)\{F\}} \quad (\text{Assert}) \\
\\
\frac{\Gamma \vdash v : T \quad \Gamma; o; F \vdash G[v/\alpha] \quad T <: U \quad \Gamma, \ell : U; o \vdash \{(\text{ex } T \alpha)(\alpha == \ell * G)\}c : V\{H\}}{\Gamma, \ell : U; o \vdash \{F\}\ell = \text{return}(v); c : V\{H\}} \quad (\text{Return})
\end{array}$$

Remarks. The rule (Assert) is defined for a specification-only `assert` statement, that is formally defined on page 17. Intuitively, `assert(G)` expresses that G should hold at that point in the execution. It is used to express a corollary about partial correctness of a verified program. The rule (Return) is for the auxiliary `return` statement, defined in Section 2.2. As explained, source code programs do not contain this statement, but we need the rule to prove soundness of the proof system.

Well-formedness. The following presents formal definitions of well-formed predicate types, method types and verified interfaces

Well-formed Predicate Types, Method Types , Verified Interfaces :

$$\frac{\Gamma \vdash \bar{T} : \diamond}{\Gamma \vdash \text{pred } P < \bar{T} \bar{\alpha} > : \diamond} \quad (\text{Pred Type}) \quad \frac{\Gamma \vdash F : \diamond}{\Gamma \vdash \text{axiom } F : \diamond} \quad (\text{Ax})$$

$$\frac{\Gamma, \bar{\alpha} : \bar{T}, \bar{\iota} : \bar{V} \vdash \bar{T}, F, U, \bar{V} : \diamond \quad \Gamma, \bar{\alpha} : \bar{T}, \bar{\iota} : \bar{V}, \text{result} : U \vdash G : \diamond}{\Gamma \vdash \langle \bar{T} \bar{\alpha} \rangle \text{requires } F; \text{ensures } G; U \text{ m}(\bar{V} \bar{\iota}) : \diamond} \quad (\text{Mth Type})$$

$$\frac{I \langle \bar{T} \bar{\alpha} \rangle \text{type-extends } \bar{U} \quad \text{init} \notin \text{dom}(pt^*) \quad \bar{\alpha} : \bar{T} \vdash \bar{T}, \bar{U}, pt^* : \diamond \quad \bar{\alpha} : \bar{T}, \text{this} : I \langle \bar{\alpha} \rangle \vdash ax, mt^* : \diamond}{\text{interface } I \langle \bar{T} \bar{\alpha} \rangle \text{ext } \bar{U} \{pt^* \ ax^* \ mt^*\} : \diamond} \quad (\text{Int})$$

We write $\text{cfv}(c)$ for the set of variables that occur freely in an object creation command in c . Rule (Cls) below is the main judgment for verifying classes. Premises $C \langle \bar{T} \bar{\alpha} \rangle \text{ext } U$ and $C \langle \bar{T} \bar{\alpha} \rangle \text{impl } \bar{V}$ enforce class C to be sane. Premise $C \langle \bar{T} \bar{\alpha} \rangle \text{sound}$ enforces C 's axioms to be sound. Premise $\bar{\alpha} : \bar{T}, \text{this} : C \langle \bar{\alpha} \rangle \vdash fd^*, ax^*, md^* : \diamond$ enforces C 's methods (md^*) to be verified.

Rule (Mth) below verifies methods. In this rule, we prohibit object creation commands to contain logical method parameters because our operational semantics does not keep track of logical method parameters (while it does keep track of class parameters).

Verified Classes, $cl : \diamond$:

$$\frac{C \langle \bar{T} \bar{\alpha} \rangle \text{ext } U \quad C \langle \bar{T} \bar{\alpha} \rangle \text{impl } \bar{V} \quad C \langle \bar{T} \bar{\alpha} \rangle \text{sound} \quad \text{init} \notin \text{dom}(pd^*) \quad \bar{\alpha} : \bar{T} \vdash \bar{T}, U, \bar{V} : \diamond \quad \bar{\alpha} : \bar{T} \vdash pd^* : \diamond \text{ in } C \langle \bar{\alpha} \rangle \quad \bar{\alpha} : \bar{T}, \text{this} : C \langle \bar{\alpha} \rangle \vdash fd^*, ax^*, md^* : \diamond}{\text{class } C \langle \bar{T} \bar{\alpha} \rangle \text{ext } U \text{impl } \bar{V} \{fd^* \ pd^* \ ax^* \ md^*\} : \diamond} \quad (\text{Cls})$$

$$\frac{\Gamma \vdash T : \diamond}{\Gamma \vdash T f : \diamond} \quad (\text{Fld}) \quad \frac{\Gamma \vdash \text{pred } P \langle \bar{T} \bar{\alpha} \rangle : \diamond \quad \Gamma, \text{this} : U, \bar{\alpha} : \bar{T} \vdash F : \diamond}{\Gamma \vdash \text{pred } P \langle \bar{T} \bar{\alpha} \rangle = F : \diamond \text{ in } U} \quad (\text{Pred})$$

$$\frac{\Gamma \vdash \langle \bar{T} \bar{\alpha} \rangle \text{requires } F; \text{ensures } G; U \text{ m}(\bar{V} \bar{\iota}) : \diamond \quad \text{cfv}(c) \cap \bar{\alpha} = \emptyset \quad \Gamma' = \Gamma, \bar{\alpha} : \bar{T}, \bar{\iota} : \bar{V} \quad \Gamma'; \text{this} \vdash \{F^* \ \text{this} \neq \text{null}\} c : U \{(\text{result}) (G)\}}{\Gamma \vdash \langle \bar{T} \bar{\alpha} \rangle \text{requires } F; \text{ensures } G; U \text{ m}(\bar{V} \bar{\iota}) \{c\} : \diamond} \quad (\text{Mth})$$

Sanity Conditions for Class Extensions and Interface Implementations. In the definition below, we treat the partial functions mtype and ptype (formally defined in A.1) as total functions that map elements outside their domains to the special element undef . Furthermore, we extend the subtyping relation:

$$\prec := \{(T, U) \mid T \prec U\} \cup \{(\text{undef}, \text{undef})\}$$

$$C \langle \bar{T} \bar{\alpha} \rangle \text{ext } U \triangleq \begin{cases} U \text{ is a parameterized class} \\ f \in \text{dom}(\text{fld}(U)) \Rightarrow f \notin \text{declared}(C) \\ (\forall m, mt)(\text{mtype}(m, U) = mt \Rightarrow \\ \quad \bar{\alpha} : \bar{T} \vdash \text{mtype}(m, C \langle \bar{\alpha} \rangle) \prec mt) \\ (\forall P, pt)(\text{ptype}(P, U) = pt \Rightarrow \text{ptype}(P, C \langle \bar{\alpha} \rangle) \prec pt) \end{cases}$$

$$I \langle \bar{T} \bar{\alpha} \rangle \text{type-extends } U \triangleq \begin{cases} U \text{ is a (parameterized) interface} \\ (\forall m, mt)(\text{mtype}(m, U) = mt \Rightarrow \\ \quad \bar{\alpha} : \bar{T} \vdash \text{mtype}(m, I \langle \bar{\alpha} \rangle) \prec mt) \\ (\forall P, pt)(\text{ptype}(P, U) = pt \Rightarrow \\ \quad \text{ptype}(P, I \langle \bar{\alpha} \rangle) \prec pt) \end{cases}$$

$$I \langle \bar{T} \bar{\alpha} \rangle \text{type-extends } \bar{U} \triangleq (\forall U \in \bar{U})(I \langle \bar{T} \bar{\alpha} \rangle \text{type-extends } U)$$

$$\begin{aligned}
C\langle\bar{T}\ \bar{\alpha}\rangle \text{ impl } U &\triangleq \left\{ \begin{array}{l} U \text{ is a (parameterized) interface} \\ (\forall m, mt)(\text{mtype}(m, U) = mt \Rightarrow \\ \quad \text{mtype}(m, C\langle\bar{\alpha}\rangle) \neq \text{undef}) \\ (\forall P, pt)(\text{ptype}(P, U) = pt \Rightarrow \text{ptype}(P, C\langle\bar{\alpha}\rangle) \neq \text{undef}) \\ (\forall m, mt)(\text{mtype}(m, U) = mt \Rightarrow \\ \quad \bar{\alpha} : \bar{T} \vdash \text{mtype}(m, C\langle\bar{\alpha}\rangle) <: mt) \\ (\forall P, pt)(\text{ptype}(P, U) = pt \Rightarrow \text{ptype}(P, C\langle\bar{\alpha}\rangle) <: pt) \end{array} \right. \\
C\langle\bar{T}\ \bar{\alpha}\rangle \text{ impl } \bar{U} &\triangleq (\forall U \in \bar{U})(C\langle\bar{T}\ \bar{\alpha}\rangle \text{ impl } U)
\end{aligned}$$