# Permission-Based Separation Logic for Multithreaded Java Programs[☆]

Christian Haack[a,1], Marieke Huisman[b,2], Clément Hurlin[c,3,*]

[a] *Aicas GmbH, Karslruhe, Germany*
[b] *University of Twente, The Netherlands*
[c] *IRISA/Université de Rennes 1, France*

**Abstract**

This paper presents a program logic for reasoning about multithreaded Java-like programs with concurrency primitives such as 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 entrancies and final monitor exits. In order to distinguish between initial monitor entrancies and monitor reentrancies, 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 (i.e., invariants that are never invalid, not even temporarily), 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.

[☆]This work was funded in part by the 6th Framework programme of the EC under the MOBIUS project IST-FET-2005-015905.

[*]Corresponding author

[1]Part of the work done while the author was at Radboud University Nijmegen, Netherlands.

[2]Part of the work done while the author was at INRIA Sophia Antipolis – Méditerranée, France.

[3]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 and Microsoft R&D, 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 [1], static analysis tools for data race and deadlock detection [2, 3], type-and-effect systems for atomicity [4, 5], and program verification tools based on interactive theorem proving [6]. A particularly successful line of research is concerned with static contract checking tools based on Hoare logic. Examples include ESC/Java [7] — a highly automatic, but deliberately unsound, tool based on a weakest precondition calculus and an SMT solver, the Key tool [8] — a sound verification tool for Java programs based on dynamic logic and symbolic execution, and Spec# [9] — 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 their support for concurrency 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.

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 [10] or Jones's rely-guarantee method [11], 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 [12] and monitors [13] are much simpler, because they rely on syntactically enforceable synchronization disciplines that limit thread interference to few synchronization points (see [14] 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 [15] has generalized Hoare's logic to programming languages with heap. To this end, he extended a new program logic, called *separation logic* [16, 17], which had previously been used for reasoning about sequential pointer programs. O'Hearn's *concurrent separation logic (CSL)* 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. In this paper, we adapt CSL to a Java-like language.

Adapting CSL to Java requires a number of substantial extensions: Firstly, while O'Hearn's CSL assumes a static set of locks, in Java locks have the same status as other objects that are dynamically allocated and stored on the heap, and can be aliased. Secondly, while O'Hearn's CSL assumes structured parallelism[4], 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. A join-operation that is parametrized by a thread identifier allows threads to wait for the termination of other threads. Thirdly, while O'Hearn's CSL assumes that programs go through a global initialization phase to establish all invariants, this assumption is inappropriate for Java programs where objects and locks are created dynamically and, consequently, initialize their invariants dynamically. Fourthly, while O'Hearn considers classical monitors that cannot be reentered, Java's monitors are reentrant. Reentrant monitors have the advantage of avoiding 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. Fifthly, O'Hearn's CSL does not allow multiple threads to read the same location simultaneously. This is more restrictive than necessary: to avoid data races read-write and write-write conflicts must be avoided, but concurrent reads are harmless.

CSL has since been extended in various directions to overcome some of these limitations. For instance, Bornat and others have combined separation logic with permission accounting in order to support concurrent reads [19], or Gotsman and others have generalized concurrent separation logic to cope with Posix-style threads and locks [20]. This paper takes the ideas from concurrent separation logic into another direction, namely towards reasoning about multithreaded *Java-like* programs. The resulting proof system supports Java's main concurrency primitives: dynamically created threads and monitors that can be stored on the heap, thread joining, and monitor reentrancy. 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 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 [17] combines the usual logical operators with the points-to predicate $x.f \mapsto v$ and the resource conjunction $F * G$.

[4]For presentational reasons, O'Hearn's paper [15] assumes a fixed, static set of threads, but remarks that this can be generalized to structured parallelism, as done by Brookes [18].

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 $(\exists v)(x.f \mapsto v)$):

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

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 forgeable: one ticket is not the same as two tickets! For this reason, the resource conjunction $*$ is not idempotent: $F$ is not equivalent to $F * F$. 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 `a` and `b` with field `x`, the assertion $\texttt{a.x} \mapsto \_ * \texttt{b.x} \mapsto \_$ does not mean the same as $\texttt{a.x} \mapsto \_ \wedge \texttt{b.x} \mapsto \_$: the first assertion implies that `a` and `b` are distinct, while the second assertion can be satisfied even if `a` and `b` are aliases.

*1.2.2. Local Reasoning*

A crucial feature of separation logic is that it allows to reason locally about methods. This means that, when calling a method, one can identify (1) the (small) part of the heap accessed by that method and (2) the rest of the heap that is left unaffected. Formally, this is expressed by the (Frame) rule:

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

This rule expresses that given a command $c$ which 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.

*1.2.3. Abstraction*

A good object-oriented programming practice is to consider objects abstractly i.e., to hide implementation details from clients. To this end, Parkinson introduced *abstract predicates* [21]. Abstract predicates hide implementation details from clients but allow class implementers to use them. In other words, abstract predicates are *opaque* to clients but *transparent* to class implementers.

### 1.3. Contributions

Using the aspects of separation logic described above, we have developed a program logic for a concurrent language with Java's main concurrency primitives. Our logic combines separation logic with fraction-based permissions. 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 6.

Because of the use of fraction-based permission permissions, as proposed by Boyland [22], 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. Our logic supports multiple thread joiners. 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 [15], 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 on). 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. Our program logic supports reentrant locks. 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 [23] 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., [24, 25]). Similar ownership type systems have been used in program verification systems to control aliasing (e.g, [26]). In Section 5.6.2, 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 [21]. 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. We support modular verification of predicate extensions, by axiomatizing the so-called "stack of class frames" [27, 9].

### 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 [28], the logic to reason about reentrant locks was presented at APLAS 2008 [29]. 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 forms a part of Hurlin's PhD thesis [30].

The remainder of this paper is organized as follows. Section 2 presents the Java-like language that we use for our formal development. Section 3 introduces permission-based separation logic and basic proof rules for single-threaded programs. Section 4 extends this to multithreaded programs with dynamic thread creation and termination, while Section 5 adds reentrant locks. Finally, Sections 6 and 7 discuss related work, future work and conclusions. The soundness proof for the system can be found in Hurlin's PhD thesis [30].

## 2. The Java-like model language

This section presents the *sequential* Java-like language that forms the basis for this paper. Later sections will extend the language with Java-like concurrency primitives. The language models core features of Java: mutable fields, inheritance and method overriding, and interfaces. It does not model, however, *all* features of Java, omitting for instance static fields, super calls, inner classes and reflection. Modulo minor syntactic differences for the sake of presentational simplicity, the model language is a subset of Java. 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., [31, 32, 33, 34]).

### 2.1. Syntax

The language distinguishes between read-only variables $\imath$, read-write variables $\ell$, and logical variables $\alpha$. 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 3.2). In concrete Java-syntax, read-only variables can be represented by preceding their declarations with Java's `final`-modifiers, but we omit the `final`-modifiers here. The model language also includes class identifiers, interface identifiers, field identifiers, method identifiers and predicate identifiers. Object identifiers are used in the operational semantics, but must not occur in source programs.

**Identifier Domains:**

| | |
|---|---|
| $C, D \in \mathsf{ClassId}$ | class identifiers (including `Object`) |
| $I, J \in \mathsf{IntId}$ | interface identifiers |
| $s, t \in \mathsf{TypeId} = \mathsf{ClassId} \cup \mathsf{IntId}$ | type identifiers |
| $o, p, q, r \in \mathsf{ObjId}$ | object identifiers (must not occur in source programs) |
| $f \in \mathsf{FieldId}$ | field identifiers |
| $m \in \mathsf{MethId}$ | method identifiers |
| $P \in \mathsf{PredId}$ | predicate identifiers |
| $\imath \in \mathsf{RdVar}$ | read-only variables (including `this`) |
| $\ell \in \mathsf{RdWrVar}$ | read-write variables |
| $\alpha \in \mathsf{LogVar}$ | logical variables |
| $x, y, z \in \mathsf{Var} = \mathsf{RdVar} \cup \mathsf{RdWrVar} \cup \mathsf{LogVar}$ | variables |

(Open) values are integers, booleans, object identifiers, `null`, and read-only variables. *Closed values* are values that are not variables. Initially, *specifications values* range over logic variables and values; this will be extended in subsequent sections.

$$n \in \mathsf{Int} \qquad u, v, w \in \mathsf{Val} ::= \texttt{null} \mid n \mid b \mid o \mid \imath$$
$$b \in \mathsf{Bool} = \{\texttt{true}, \texttt{false}\} \qquad \mathsf{ClVal} = \mathsf{Val} \setminus \mathsf{RdVar}$$
$$\pi \in \mathsf{SpecVal} ::= \alpha \mid v$$

Now we define the *types* used in our language. Since interfaces and classes (defined next) can be parameterized with specification values, object types are of the form $t\texttt{<}\bar{\pi}\texttt{>}$.

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

*Expressions* are built from values and variables using arithmetic and logical operators. We introduce a special `instanceof` operator that is exactly Java's `instanceof`.

$$\begin{aligned} op \quad \supseteq \quad & \{\texttt{==}, \texttt{!}, \texttt{\&}, \texttt{|}\} \cup \{\, C\ \texttt{classof} \mid C \in \mathsf{ClassId} \,\} \\ & \cup \{\, \texttt{instanceof}\ T \mid T \in \mathsf{Type} \,\} \end{aligned}$$

$$e \in \mathsf{Exp} \ ::= \ \pi \ \mid \ \ell \ \mid \ op\texttt{(}\bar{e}\texttt{)}$$

Next, *class declarations* are defined. As in Java, classes can `ext`(end) other classes, and `impl`(ement) interfaces. Classes declare *fields*, *abstract predicates*, *class axioms*, and *methods*. Abstract predicates [21, 35] and class axioms are part of our specification language; they are explained in Section 3.1.1. Appendix A.1 defines syntactic functions to lookup fields (fld), axioms (axiom), method types and bodies (mtype and mbody, respectively), and predicate types and bodies (ptype and pbody, respectively). Methods have pre/postcondition specifications, parameterized by logical variables. The meaning of a specification is defined via a universal quantification over these parameters. In examples, we usually leave the parameterization implicit, but it is treated explicitly in the formal language.

**Class Declarations:**

| | |
|---|---|
| $F \in \mathsf{Formula}$ | specification formulas (see Section 3.1.1) |
| $spec \ ::= \ \texttt{requires}\ F\texttt{;}\ \texttt{ensures}\ F\texttt{;}$ | pre/postconditions |
| $fd \ ::= \ T\ f\texttt{;}$ | field declarations |
| $pd \ ::= \ \texttt{pred}\ P\texttt{<}\bar{T}\ \bar{\alpha}\texttt{>} = F\texttt{;}$ | predicate definitions (see Section 3.1.1) |
| $ax \ ::= \ \texttt{axiom}\ F\texttt{;}$ | class axioms (see Section 3.1.1) |
| $md \ ::= \ \texttt{<}\bar{T}\ \bar{\alpha}\texttt{>}\ spec\ U\ m\texttt{(}\bar{V}\ \bar{\imath}\texttt{)\{}c\texttt{\}}$ | methods (scope of $\bar{\alpha}, \bar{\imath}$ is $\bar{T}, spec, U, \bar{V}, c$) |
| $cl \in \mathsf{Class} \ ::= \ \texttt{class}\ C\texttt{<}\bar{T}\ \bar{\alpha}\texttt{>}\ \texttt{ext}\ U\ \texttt{impl}\ \bar{V}\ \texttt{\{}fd\texttt{*}\ pd\texttt{*}\ ax\texttt{*}\ md\texttt{*\}}$ | class (scope of $\bar{\alpha}$ is $\bar{T}, U, \bar{V}, fd\texttt{*}, pd\texttt{*}, ax\texttt{*}, md\texttt{*}$) |

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`.

In a similar way, we define *interfaces*:

**Interface Declarations:**

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

Class and interface declarations define *class tables*: $ct \subseteq \mathsf{Interface} \cup \mathsf{Class}$. We use the symbol $\preceq_{ct}$ for the order on type identifiers induced by class table

$ct$. We often leave the subscript $ct$ implicit. We impose the following sanity conditions on $ct$: (1) $\preceq_{ct}$ is antisymmetric, (2) if $t$ (except `Object`) occurs anywhere in $ct$ then $t$ is declared in $ct$ and (3) $ct$ does not contain duplicate declarations or a declaration of `Object`. These sanity conditions are the same as in Featherweight Java [36]. We write $\mathsf{dom}(ct)$ for the set of all type identifiers declared in $ct$.

*Subtyping* is inductively defined by the following rules:

**Subtyping $T <: T$:**

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

Commands are sequences of head commands $hc$ and local variable declarations, terminated by a return value:

**Commands:**

$$\begin{array}{rcl} c \in \mathsf{Cmd} & ::= & v \mid T\ \ell; c \mid T\ \imath = \ell; c \mid hc; c \\ hc \in \mathsf{HeadCmd} & ::= & \ell = v \mid \ell = op(\bar{v}) \mid \ell = v.f \mid v.f = v \mid \ell = \texttt{new}\ C\texttt{<}\bar{\pi}\texttt{>} \mid \\ & & \ell = v.m\texttt{(}\bar{v}\texttt{)} \mid \texttt{if (}v\texttt{)\{}c\texttt{\}else\{}c\texttt{\}} \mid \texttt{while (}e\texttt{)\{}c\texttt{\}} \end{array}$$

The meaning of the Java commands is exactly as in Java. 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. Hurlin's thesis [30] illustrates how a Java program is written precisely in our language.

### 2.2. Semantics

This section describes the small-step operational semantics of our Java-like language. This semantics 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. We choose to model parameter-passing by substitution. Local read-write variables are mapped to closed values by a single mathematical function (which we name *stack*), relying on alpha-conversion to avoid naming conflicts. In this way, we avoid modeling the details of a call stack.

*Runtime structures.* The small-step operational semantics operates on states, consisting of a heap, a command, and a stack. Section 4 will extend the state, to cope with multithreaded programs. As usual, a *heap* maps each object identifier (1) to its dynamic type and (2) to a mapping from fields to closed values:

$$\mathsf{ObjStore} = \mathsf{FieldId} \rightharpoonup \mathsf{ClVal} \qquad h \in \mathsf{Heap} = \mathsf{ObjId} \rightharpoonup \mathsf{Type} \times \mathsf{ObjStore}$$

Given a heap $h$ and an object identifier $o$, we write $h(o)_1$ to denote $o$'s dynamic type and $h(o)_2$ to denote $o$'s store.

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

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

Finally, a *state* consists of a heap, a command, and a stack:

$$st \in \mathsf{State} = \mathsf{Heap} \times \mathsf{Cmd} \times \mathsf{Stack}$$

*Initialization.* To define initial states, we first define functions to initialize objects and programs.

Function $\mathsf{df} : \mathsf{Type} \to \mathsf{ClVal}$ maps types to their default values:

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

Function $\mathsf{initStore} : \mathsf{Type} \rightharpoonup \mathsf{ObjStore}$ maps object types to their initial object stores.

$$\mathsf{initStore}(t\texttt{<}\bar{\pi}\texttt{>})(f) = \mathsf{df}(T) \text{ iff } (T\ f) \in \mathsf{fld}(C\texttt{<}\pi\texttt{>})$$

Now, function $\mathsf{init} : \mathsf{Cmd} \to \mathsf{State}$ maps programs to their initial state. Initially, the heap is empty (hence the first $\emptyset$) and the stack is empty (hence the second $\emptyset$):

$$\mathsf{init}(c) = \langle \emptyset, c, \emptyset \rangle$$

*Semantics of values.* The semantics of read-only variables is left undefined, because we deal with them by substitution. For values that are not read-only variables, their semantics is simply identity:

**Semantics of Values, $[\![v]\!] \in \mathsf{ClVal}$:**

$$[\![\texttt{null}]\!] \triangleq \texttt{null} \quad [\![n]\!] \triangleq n \quad [\![b]\!] \triangleq b \quad [\![o]\!] \triangleq o$$

*Semantics of operators.* We require that the built-in operators include reference equality `==`, boolean negation `!`, boolean conjunction `&`, and boolean disjunction `|`. Furthermore, we require that for each class $C$, there is an operator '$C$ `classof` $v$' that tests whether $C$ is $v$'s dynamic class. Note that this last operator depends on object types, as stored on the heap. Another example of an operator that depends on object types is our `instanceof`. The formal semantics of the built-in operators are presented in Appendix A.2.

In addition to the above operators, we permit to use any further built-in operator that satisfies the following two axioms:

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

The first of these axioms ensures that operators are invariant under heap extensions. The second axiom ensures that operators do no depend on values stored on the heap. Thus, if operators depend on the heap at all, then they may only depend on types (but not on values) stored on the heap. This is useful for the operators `classof` and `instanceof` that depend on the type components of objects. Because it is fixed at object creation, these operators satisfy requirement (b).

*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 ClVal; but it will be extended in Sec. 3.1.4 as we extend specification values. We range over SemVal with meta-variable $\mu$.

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

(Sem SpecVal) $\dfrac{[\![\pi]\!] = \mu}{[\![\pi]\!]^h_s = \mu}$ (Sem Var) $\dfrac{s(\ell) = v}{[\![\ell]\!]^h_s = v}$ (Sem Op) $\dfrac{[\![w_1]\!]^h_s = v_1 \ \cdots \ [\![w_n]\!]^h_s = v_n \quad [\![op]\!]^h(v_1, \ldots, v_n) = v}{[\![op(w_1, \ldots, w_n)]\!]^h_s = v}$

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

*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 $\ell$. In its definition, we write $\mathsf{fv}(c)$ for the set of free variables of $c$. Furthermore, we make use of a piece of auxiliary syntax $\ell\,\texttt{=return(}v\texttt{);}\ c$. This construct is not meant to be used in source programs. Its purpose is to mark method-return-points in intermediate program states. Tracking method-return-points in this way will be needed to show soundness of the proof system.

$$
\begin{array}{rcll}
\ell \leftarrow v;\ c & \triangleq & \ell\,\texttt{=return(}v\texttt{);}\ c & \\
\ell \leftarrow (T\ \ell';\ c);\ c' & \triangleq & T\ \ell';\ \ell \leftarrow c;\ c' & \text{if } \ell' \notin \mathsf{fv}(c') \text{ and } \ell' \neq \ell \\
\ell \leftarrow (T\ \imath\,\texttt{=}\,\ell';\ c);\ c' & \triangleq & T\ \imath\,\texttt{=}\,\ell';\ \ell \leftarrow c;\ c' & \text{if } \imath \notin \mathsf{fv}(c') \\
\ell \leftarrow (hc;\ c);\ c' & \triangleq & hc;\ \ell \leftarrow c;\ c' &
\end{array}
$$

$c\ ::=\ \ldots\ \mid\ \ell\,\texttt{=return(}v\texttt{);}\ c\ \mid\ \ldots$
*Restriction:* This clause must not occur in source programs.

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

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

*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 $ct$. In the reduction rules, we use the following abbreviation for field updates: $h[o.f \mapsto v] = h[o \mapsto (h(o)_1, h(o)_2[f \mapsto v])]$.

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

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

(Red Fin Dcl) $s(\ell) = v \quad c' = c[v/\imath]$
$\langle h,\ T\ \imath\,\texttt{=}\,\ell;\ c, s\rangle \ \rightarrow\ \langle h,\ c', s\rangle$

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

(Red Op) $\mathsf{arity}(op) = |\bar{v}| \quad [\![op]\!]^h(\bar{v}) = w \quad s' = s[\ell \mapsto w]$
$\langle h,\ \ell = op(\bar{v});\ c, s\rangle \ \rightarrow\ \langle h,\ c, s'\rangle$

(Red Get) $s' = s[\ell \mapsto h(o)_2(f)]$
$\langle h,\ \ell = o.f;\ c, s\rangle \ \rightarrow\ \langle h,\ c, s'\rangle$

(Red Set) $h' = h[o.f \mapsto v]$
$\langle h,\ o.f = v;\ c, s\rangle \ \rightarrow\ \langle h',\ c, s\rangle$

(Red New) $o \notin \mathsf{dom}(h) \quad h' = h[o \mapsto (C\texttt{<}\bar{\pi}\texttt{>}, \mathsf{initStore}(C\texttt{<}\bar{\pi}\texttt{>}))] \quad s' = s[\ell \mapsto o]$
$\langle h,\ \ell = \texttt{new}\ C\texttt{<}\bar{\pi}\texttt{>};\ c, s\rangle \ \rightarrow\ \langle h',\ c, s'\rangle$

(Red Call) $h(o)_1 = C\texttt{<}\bar{\pi}\texttt{>} \quad \mathsf{mbody}(m, C\texttt{<}\bar{\pi}\texttt{>}) = (\imath_0; \bar{\imath}).c_m \quad c' = c_m[o/\imath_0, \bar{v}/\bar{\imath}]$
$\langle h,\ \ell = o.m\texttt{(}\bar{v}\texttt{)};\ c, s\rangle \ \rightarrow\ \langle h,\ \ell \leftarrow c';\ c, s\rangle$

(Red Return)
$\langle h,\ \ell = \texttt{return(}v\texttt{)};\ c, s\rangle \ \rightarrow\ \langle h,\ \ell = v;\ c, s\rangle$

(Red If True)
$\langle h,\ \texttt{if (true)\{}c\texttt{\}else\{}c'\texttt{\}};\ c'', s\rangle \ \rightarrow\ \langle h,\ c;\ c'', s\rangle$

(Red If False)
$\langle h,\ \texttt{if (false)\{}c\texttt{\}else\{}c'\texttt{\}};\ c'', s\rangle \ \rightarrow\ \langle h,\ c';\ c'', s\rangle$

(Red While True) $[\![e]\!]^h_s = \texttt{true}$
$\langle h,\ \texttt{while (}e\texttt{)\{}c\texttt{\}};c', s\rangle \ \rightarrow\ \langle h,\ c;\ \texttt{while (}e\texttt{)\{}c\texttt{\}};c', s\rangle$

(Red While False) $[\![e]\!]^h_s = \texttt{false}$
$\langle h,\ \texttt{while (}e\texttt{)\{}c\texttt{\}};c', s\rangle \ \rightarrow\ \langle h,\ c', s\rangle$

*Remarks.* 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), $\imath_0$ is the formal method receiver and $\bar{\imath}$ 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.

## 3. Separation Logic for a Java-like Language

This section shows how sequential programs written in the Java-like language from Section 2 can be specified and verified with separation logic. The next two sections will extend this to a multithreaded language. This section first presents separation logic formally, then it presents the proof system and shows soundness of the approach. As an example, we show the specification of a sequential merge sort algorithm.

### 3.1. Separation Logic

*3.1.1. Syntax*

To write method contracts, we use *intuitionistic* separation logic [16, 17, 21]. This is most suitable to reason about properties that are invariant under heap extensions, and to reason about garbage-collected languages like Java. Contrary to classical separation logic, intuitionistic separation logic admits weakening.

Informally, this means that one can "forget" a part of the state, which makes it appropriate for garbage-collected languages.

*Specification formulas* $F$ are defined by the following grammar:

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

We now explain these formulas:

The *points-to predicate* $\texttt{PointsTo(}e.f, \pi, v\texttt{)}$ is ASCII for $e.f \overset{\pi}{\longmapsto} v$ [19]. Superscript $\pi$ must be a fractional permission [22] i.e., a fraction $\frac{1}{2^n}$ in the interval $(0, 1]$. As explained earlier, formula $\texttt{PointsTo(}e.f, \pi, v\texttt{)}$ has a dual meaning: firstly, it asserts that field $e.f$ contains value $v$, and, secondly, it represents access right $\pi$ to $e.f$. Permission $\pi = 1$ grants write access while any permission $\pi$ grants read access.

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

The *resource implication* $F \texttt{-*} G$ (a.k.a. *linear implication* or *magic wand*) means "consume $F$ yielding $G$". Resource $F \texttt{-*} G$ permits to trade resource $F$ to receive resource $G$ in return. Resource conjunction and implication are related by the modus ponens: $F \texttt{*} \texttt{(}F \texttt{-*} G\texttt{)}$ implies $G$. Most related work omit the magic wand. We include it, because it can be added without any difficulties, and we found it useful to specify some programming patterns. In particular our previous work [23] exemplifies contracts that make heavy use of it.

To avoid a proof theory with bunched contexts (see Section 3.1.5), we omit the $\Rightarrow$-implication between heap formulas (and did not need it in later examples).

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

For expressivity, it is crucial to allow parameterization of objects by permissions. For this, we include a special type `perm` for fractional permissions:

$$T, U, V, W \in \mathsf{Type} \quad ::= \quad \ldots \mid \texttt{perm} \mid \ldots$$

Because class parameters are instantiated by specification values (see Section 2.1), we extend specification values with fractional permissions. Fractional

permissions are represented symbolically: 1 represents itself, and if symbolic fraction $\pi$ represents concrete fraction $fr$ then $\texttt{split}(\pi)$ represents $\frac{1}{2} \cdot fr$.

$$\pi \in \mathsf{SpecVal} \quad ::= \quad \ldots \mid 1 \mid \texttt{split}(\pi) \mid \ldots$$

Quantified formulas have the shape $\texttt{(}qt\ T\ \alpha\texttt{)(}F\texttt{)}$, where $qt$ is a universal or existential quantifier, $\alpha$ is a variable whose scope is formula $F$, and $T$ is $\alpha$'s type. Because specification values $\pi$ and expressions $e$ may contain logical variables $\alpha$ (see pages 7 and 8), quantified variables can appear in many positions: as type parameters; as the first, third, and fourth parameter in `PointsTo` predicates [5]; as predicate parameters etc. This is important for expressiveness.

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

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

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

In our model language, specifications are expressed with method pre- and postconditions. Preconditions are declared with keyword `requires`, while postconditions are declared with keyword `ensures`. In postconditions, the special identifier `result` can be used to refer to the value returned. As defined in Section 2.1, the syntax of a method declaration with pre- and postconditions is as follows:

$$\texttt{<}\bar{T}\ \bar{\alpha}\texttt{>requires}\ F\texttt{;ensures}\ F\texttt{;}\ U\ m\texttt{(}\bar{V}\ \bar{v}\texttt{)\{}c\texttt{\}}$$

Interfaces may declare *abstract predicates* and classes may implement them by providing concrete definitions as separation logic formulas.

$$\texttt{interface I \{ ... pred }\ P\texttt{<}\bar{T}\ \bar{x}\texttt{>; ... \}}$$
$$\texttt{class C impl I \{ ... pred }\ P\texttt{<}\bar{T}\ \bar{x}\texttt{>=}F\texttt{; ... \}}$$

Class axioms export facts about relations between abstract predicates, without revealing the detailed predicate implementations. Class implementors have to prove class axioms and class clients can use them.

$$\texttt{class C \{ ... axiom }\ F\texttt{;... \}}$$

[5] Note 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.

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 3.5). Formally, `public` desugars to an `axiom` (where $C$ is the enclosing class):

$$\texttt{public pred } P\texttt{<}\bar{T}\,\bar{x}\texttt{>} = F \quad \triangleq \quad \begin{array}{l}\texttt{pred } P\texttt{<}\bar{T}\,\bar{x}\texttt{>} = F\texttt{;}\\ \texttt{axiom } P\texttt{@}C\texttt{<}\bar{x}\texttt{>} \texttt{ *-* } F\end{array}$$

Notice that `public pred P<`$\bar{T}\,\bar{x}$`> = `$F$`;` is not equivalent to `pred P<`$\bar{T}\,\bar{x}$`> =` $F$`;` and `axiom P<`$\bar{T}\,\bar{x}$`> *-*` $F$`;`. The latter axiom is more general than `public`'s desugaring because it constrains extensions of `P` in $C$'s subclasses to satisfy this axiom (whereas `public`'s desugaring simply exports `P`'s definition *in class* $C$). For additional usage of `public`, we refer to [30, §3.2.1] and Sections 4.5 and 5.6.

*3.1.2. Resources*

This paragraph defines *resources* i.e., models of our formulas. These will be used below, in Section 3.1.4, when we define a forcing relation of the form $\Gamma \vdash \mathcal{E}; \mathcal{R}; s \models F$, where $\Gamma$ is a *type environment*, $\mathcal{E}$ is a *predicate environment* (that maps predicate identifiers to concrete heap predicates), $\mathcal{R}$ is a *resource*, and $s$ is a *stack*. Intuitively, if $\Gamma \vdash \mathcal{E}; \mathcal{R}; s \models F$ holds, resource $\mathcal{R}$ is a state that is described by $F$.

Resources $\mathcal{R}$ range over the set $\mathsf{Resource}$ with a binary relation $\# \subseteq \mathsf{Resource} \times \mathsf{Resource}$ (the *compatibility relation*) and a partial binary operator $* : \# \rightharpoonup \mathsf{Resource}$ (the *resource joining operator*) that is associative and commutative. Concretely, resources are pairs $\mathcal{R} = (h, \mathcal{P})$ of a *heap* $h$ and a *permission table* $\mathcal{P} \in \mathsf{ObjId} \times \mathsf{FieldId} \rightharpoonup [0, 1]$. We require that resources satisfy the following axioms:

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

Axiom (a) ensures that the (partial) heap $h$ only contains cells that are associated with strictly positive permissions. Technically, this condition is needed to prove soundness of the verification rule for field updates. Axiom (b) ensures that all objects that are not yet allocated have minimal permissions (with respect to the resource order presented below). This is needed to prove soundness of the verification rule for allocating new objects.

Each of the two resource components carries itself a resource structure $(\#, *)$. These structures are lifted to resources in a component-wise way. We now define $\#$ and $*$ for the two components.

*Heaps* (which, as defined in Section 2.2, have type $\mathsf{ObjId} \rightharpoonup \mathsf{Type} \times (\mathsf{FieldId} \rightharpoonup \mathsf{ClVal})$) are compatible if they agree on shared object types and memory content:

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

For example, given a class $\mathtt{C}$ with fields $\mathtt{x}$ and $\mathtt{y}$, the heaps $o \mapsto (\mathtt{C}, (\mathtt{x} \mapsto \mathtt{0}))$ and $o \mapsto (\mathtt{C}, (\mathtt{x} \mapsto \mathtt{1}))$ are incompatible, because they contain different values in field $\mathtt{x}$ of object $o$. However, heap $o \mapsto (\mathtt{C}, (\mathtt{x} \mapsto \mathtt{0}))$ and heap $o \mapsto (\mathtt{C}, (\mathtt{y} \mapsto \mathtt{1}))$ are compatible, because they contain different values in *different* fields of object $o$. Note that in this example, the two heaps each contain a field of the single object $o$: we allow partial objects.

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

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

Joining *permission tables* is defined by point-wise addition, where compatibility ensures that the sums never exceed 1.

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

For later convenience, we define projection operators $_{\mathsf{hp}}$ and $_{\mathsf{perm}}$ as follows:

$$(h, \mathcal{P})_{\mathsf{hp}} \triangleq h \qquad (h, \mathcal{P})_{\mathsf{perm}} \triangleq \mathcal{P}$$

We define an order on heaps, permission tables, and resources as follows:

$$\begin{array}{rcl} h \leq h' & \triangleq & (\exists h'')(h * h'' = h') \\ \mathcal{P} \leq \mathcal{P}' & \triangleq & (\exists \mathcal{P}'')(\mathcal{P} * \mathcal{P}'' = \mathcal{P}') \\ \mathcal{R} \leq \mathcal{R}' & \triangleq & (\exists \mathcal{R}'')(\mathcal{R} * \mathcal{R}'' = \mathcal{R}') \end{array}$$

Thus, a heap $h$ is less than a heap $h'$ if $h$ contains less memory cells than $h'$, a permission table $\mathcal{P}$ is less than a permission table $\mathcal{P}'$ if $\mathcal{P}$'s permissions are less than $\mathcal{P}'$'s permissions, and a resource $\mathcal{R}$ is less than a resource $\mathcal{R}'$ if $\mathcal{R}$'s components are all less than $\mathcal{R}'$'s components.

*3.1.3. Predicate Environments*

The predicates that are declared in the class table define a predicate environment that maps predicate symbols to relations.

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

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

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

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

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

(a) $\mathsf{fst}(\mathcal{R}_{\mathsf{hp}}) \vdash r : C\texttt{<}\bar{\pi}\texttt{>}$[6].
(b) $\mathsf{ptype}(P, C\texttt{<}\bar{\pi}\texttt{>}) = \texttt{pred}\ P\texttt{<}\bar{T}\ \bar{\alpha}\texttt{>}$ and $\mathsf{fst}(\mathcal{R}_{\mathsf{hp}}) \vdash \bar{\pi}' : \bar{T}$ for some $\bar{T}, \bar{\alpha}$.

Intuitively, if $(\bar{\pi}, \mathcal{R}, r, \bar{\pi}') \in \mathsf{Dom}(P@C)$, then $\bar{\pi}$ represents the class parameters of $r$'s dynamic class $C$, resource $\mathcal{R}$ represents the model used in the semantics relation (see Section 3.1.4), $r$ represents the predicate receiver, and $\bar{\pi}'$ represents $P$'s actual predicate parameters.

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

(a) If $(\bar{\pi}, \mathcal{R}, r, \bar{\pi}'), (\bar{\pi}, \mathcal{R}', r, \bar{\pi}') \in \mathsf{Dom}(\kappa)$ and $\mathcal{R} \leq \mathcal{R}'$, then $\mathcal{E}(\kappa)(\bar{\pi}, \mathcal{R}, r, \bar{\pi}') \leq \mathcal{E}(\kappa)(\bar{\pi}, \mathcal{R}', r, \bar{\pi}')$.

Axiom (a) says that predicates are monotone in the resources: if a predicate is satisfied in resource $\mathcal{R}$, then it is also satisfied in all larger resources $\mathcal{R}'$. This axiom is natural for a language with garbage collection. As we extend the set of formulas in next chapters, we will extend the list of axioms that predicate environments must satisfy.

*The class table's predicate environment.* The class table $ct$ defines a predicate environment that maps each predicate in $ct$ to its definition. Technically, this predicate environment is defined as the least fixed point of the endofunction[7] $\mathcal{F}_{ct}$ on predicate environments. The definition of $\mathcal{F}_{ct}$ refers to the Kripke resource semantics $\models$, as defined in Section 3.1.4.

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

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

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

**Proof.** This is a consequence of Theorem 8 on fixed points in [37]. In the proof of this theorem, it is crucial to remark that, by syntactic restriction, cyclic predicate dependencies must be positive. □

[6] All typing judgments are defined in Appendix A.3.
[7] An endofunction is a function whose range is a subset of its domain.

*3.1.4. Semantics*

Because expressions (that contain values and specification values) of type `bool` are included in the domain of formulas, the semantics of formulas depends on the semantics of expressions. We defined the semantics of expressions in Sec. 2.2. However, since we extended specification values with fractional permissions, we have to extend the semantics of specification values. Let interval $(0, 1]$ be the semantic domain of fractional permissions. The semantic domain of specification values becomes $\mathsf{SemVal} = \mathsf{ClVal} \cup (0, 1]$.

**Semantics of Specification Values, $[\![.]\!] : \mathsf{SpecVal} \to \mathsf{SemVal}$:**

$$\ldots \quad [\![1]\!] \triangleq 1 \quad [\![\texttt{split}(\pi)]\!] \triangleq \tfrac{1}{2} \cdot [\![\pi]\!] \quad \ldots$$

Next, we define the semantics of formulas. Let $(\Gamma \vdash \mathcal{R} : \diamond)$ whenever $\Gamma \vdash \mathcal{R}_{\mathsf{hp}} : \diamond$ and $\mathcal{P}(o, f) > 0$ implies $o \in \mathsf{dom}(\Gamma)$. Furthermore, let $(\Gamma \vdash \mathcal{E}, \mathcal{R}, s, F : \diamond)$ whenever $\mathcal{F}_{ct}(\mathcal{E}) = \mathcal{E}$, $\Gamma \vdash \mathcal{R} : \diamond$, $\Gamma \vdash s : \diamond$, and $\Gamma \vdash F : \diamond$[8]. The relation $(\Gamma \vdash \mathcal{E}; \mathcal{R}; s \models F)$ is the unique subset of $(\Gamma \vdash \mathcal{E}, \mathcal{R}, s, F : \diamond)$ that satisfies the following clauses:

$$
\begin{array}{rcll}
\Gamma \vdash (h, \mathcal{P}); s & \models & e & \text{iff } [\![e]\!]^h_s = \texttt{true} \\
\Gamma \vdash \mathcal{E}; (h, \mathcal{P}); s & \models & \texttt{PointsTo}(e.f, \pi, e') & \text{iff } \left\{ \begin{array}{l} [\![e]\!]^h_s = o, h(o)_2(f) = [\![e']\!]^h_s, \\ \text{and } [\![\pi]\!] \leq \mathcal{P}(o, f) \end{array} \right. \\
\Gamma \vdash \mathcal{E}; \mathcal{R}; s & \models & \texttt{null}.\kappa\texttt{<}\bar{\pi}\texttt{>} & \text{iff true} \\
\Gamma \vdash \mathcal{E}; \mathcal{R}; s & \models & o.P\texttt{@}C\texttt{<}\bar{\pi}\texttt{>} & \text{iff } \left\{ \begin{array}{l} \mathcal{R}_{\mathsf{hp}}(o)_1 <: C\texttt{<}\bar{\pi}'\texttt{>} \text{ and} \\ \mathcal{E}(P\texttt{@}C)(\bar{\pi}', \mathcal{R}, o, \bar{\pi}) = 1 \end{array} \right. \\
\Gamma \vdash \mathcal{E}; \mathcal{R}; s & \models & o.P\texttt{<}\bar{\pi}\texttt{>} & \text{iff } \left\{ \begin{array}{l} (\exists \bar{\pi}'')(\mathcal{R}_{\mathsf{hp}}(o)_1 = C\texttt{<}\bar{\pi}'\texttt{>} \text{ and} \\ \mathcal{E}(P\texttt{@}C)(\bar{\pi}', \mathcal{R}, o, (\bar{\pi}, \bar{\pi}'')) = 1) \end{array} \right. \\
\Gamma \vdash \mathcal{E}; \mathcal{R}; s & \models & F * G & \text{iff } \left\{ \begin{array}{l} (\exists \mathcal{R}_1, \mathcal{R}_2)(\mathcal{R} = \mathcal{R}_1 * \mathcal{R}_2, \\ \Gamma \vdash \mathcal{E}; \mathcal{R}_1; s \models F \text{ and} \\ \Gamma \vdash \mathcal{E}; \mathcal{R}_2; s \models G) \end{array} \right. \\
\Gamma \vdash \mathcal{E}; \mathcal{R}; s & \models & F \mathrel{-\!\!*} G & \text{iff } \left\{ \begin{array}{l} (\forall \Gamma' \supseteq_{\mathsf{hp}} \Gamma, \mathcal{R}')( \\ \mathcal{R} \# \mathcal{R}' \text{ and } \Gamma' \vdash \mathcal{E}; \mathcal{R}'; s \models F \\ \Rightarrow \ \Gamma' \vdash \mathcal{E}; \mathcal{R} * \mathcal{R}'; s \models G \ ) \end{array} \right. \\
\Gamma \vdash \mathcal{E}; \mathcal{R}; s & \models & F \,\&\, G & \text{iff } \Gamma \vdash \mathcal{E}; \mathcal{R}; s \models F \text{ and } \Gamma \vdash \mathcal{E}; \mathcal{R}; s \models G \\
\Gamma \vdash \mathcal{E}; \mathcal{R}; s & \models & F \mid G & \text{iff } \Gamma \vdash \mathcal{E}; \mathcal{R}; s \models F \text{ or } \Gamma \vdash \mathcal{E}; \mathcal{R}; s \models G \\
\Gamma \vdash \mathcal{E}; \mathcal{R}; s & \models & (\texttt{ex}\ T\ \alpha)(F) & \text{iff } \left\{ \begin{array}{l} (\exists \pi)(\ \Gamma_{\mathsf{hp}} \vdash \pi : T \text{ and} \\ \Gamma \vdash \mathcal{E}; \mathcal{R}; s \models F[\pi/\alpha] \ ) \end{array} \right. \\
\Gamma \vdash \mathcal{E}; \mathcal{R}; s & \models & (\texttt{fa}\ T\ \alpha)(F) & \text{iff } \left\{ \begin{array}{l} (\forall \Gamma' \supseteq_{\mathsf{hp}} \Gamma, \mathcal{R}' \geq \mathcal{R}, \pi)( \\ \Gamma'_{\mathsf{hp}} \vdash \mathcal{R}'_{\mathsf{hp}} : \diamond \text{ and } \Gamma'_{\mathsf{hp}} \vdash \pi : T \\ \Rightarrow \ \Gamma' \vdash \mathcal{E}; \mathcal{R}'; s \models F[\pi/\alpha] \ ) \end{array} \right.
\end{array}
$$

[8] All typing judgments are defined in Appendix A.3.

*3.1.5. Proof Theory*

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

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

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

*Semantic Validity of Boolean Expressions.* The proof theory depends on the relation $\Gamma \models e$ ("$e$ is valid in all well-typed heaps"), which we do not axiomatize (in an implementation, we would use external and dedicated theorem provers to decide this relation). To define this relation, let $\sigma$ range over *closing substitutions*, i.e, elements of $\mathsf{Var} \rightharpoonup \mathsf{ClVal}$.

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

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

We say that a heap $h$ is *total* iff for all $o$ in $\mathsf{dom}(h)$ and all $f \in \mathsf{dom}(\mathsf{fld}(h(o)_1))$ it is the case that $f \in \mathsf{dom}(h(o)_2)$ (remember that heaps were defined on page 9).

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

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

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

*Natural Deduction Rules.* The logical consequence judgment of our Hoare logic is based on the natural deduction calculus of *(affine) linear logic* [38], which coincides with BI's natural deduction calculus [39] on our restricted set of logical operators.

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

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

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

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

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

$$\frac{\Gamma; v; \bar{F}, G_1 \vdash G_2}{\Gamma; v; \bar{F} \vdash G_1 \mathbin{-\!\!*} G_2}\ (\mathbin{-\!\!*}\text{ Intro}) \qquad \frac{\Gamma; v; \bar{F} \vdash H_1 \mathbin{-\!\!*} H_2 \quad \Gamma; v; \bar{G} \vdash H_1}{\Gamma; v; \bar{F}, \bar{G} \vdash H_2}\ (\mathbin{-\!\!*}\text{ Elim})$$

$$\frac{\Gamma; v; \bar{F} \vdash G_1 \quad \Gamma; v; \bar{F} \vdash G_2}{\Gamma; v; \bar{F} \vdash G_1 \,\&\, G_2}\ (\&\text{ Intro}) \qquad \frac{\Gamma; v; \bar{F} \vdash G_1 \,\&\, G_2}{\Gamma; v; \bar{F} \vdash G_1}\ (\&\text{ Elim 1}) \qquad \frac{\Gamma; v; \bar{F} \vdash G_1 \,\&\, G_2}{\Gamma; v; \bar{F} \vdash G_2}\ (\&\text{ Elim 2})$$

$$\frac{\Gamma; v; \bar{F} \vdash G_1}{\Gamma; v; \bar{F} \vdash G_1 \mid G_2}\ (\mid\text{ Intro 1}) \qquad \frac{\Gamma; v; \bar{F} \vdash G_2}{\Gamma; v; \bar{F} \vdash G_1 \mid G_2}\ (\mid\text{ Intro 2}) \qquad \frac{\Gamma; v; \bar{F} \vdash G_1 \mid G_2 \quad \Gamma; v; \bar{E}, G_1 \vdash H \quad \Gamma; v; \bar{E}, G_2 \vdash H}{\Gamma; v; \bar{F}, \bar{E} \vdash H}\ (\mid\text{ Elim})$$

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

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

*Axioms.* In addition to the logical consequence defined above, sound *axioms* capture additional properties of our model. By axioms, we mean that they can be added to our logical consequence judgment without harming soundness, as is shown by Theorem 2 below. Table 1 presents the different axioms that we use.

The axiom (Split/Merge) regulates permission accounting (where $v$ denotes the current receiver and $\frac{\pi}{2}$ abbreviates $\texttt{split}(\pi)$).

Next, axiom (Open/Close) allows predicate receivers to toggle between predicate names and predicate definitions (where – as defined in Appendix A.1 – $\texttt{pbody}(o.P\texttt{<}\bar{\pi}'\texttt{>}, C\texttt{<}\bar{\pi}\texttt{>})$ looks up $o.P\texttt{<}\bar{\pi}'\texttt{>}$'s definition in the type $C\texttt{<}\bar{\pi}\texttt{>}$ and returns its body $F$ together with $C\texttt{<}\bar{\pi}\texttt{>}$'s direct superclass $D\texttt{<}\bar{\pi}''\texttt{>}$): 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 also that $P\texttt{@}C$ may have more parameters than $P\texttt{@}D$: following Parkinson [21] we allow subclasses to extend predicate arities. Missing predicate parameters are existentially quantified, as expressed by following axiom (Missing Parameters).

The axiom (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 (Open/Close), this allows to open and close predicates at the receiver's static type. The (`ispartof` Monotonic) axiom is similar. The axiom (Known Type) allows to drop the class modifier $C$ from $\pi.P\texttt{@}C$ if we know that $C$ is $\pi$'s dynamic class.

The next axioms define the semantics of predicates with `null`-receiver, and of `true` and `false`. The (Substitutivity) axiom allows to replace expressions by equal expressions, while (Semantic Validity) lifts semantic validity of boolean expressions to the proof theory.

The axiom (Unique Value) captures that fields point to a unique value.

$$\Gamma; v \vdash \texttt{PointsTo}(e.f, \pi, e') \mathrel{*\!\!-\!\!*} \left(\begin{array}{c} \texttt{PointsTo}(e.f, \frac{\pi}{2}, e') \\ * \\ \texttt{PointsTo}(e.f, \frac{\pi}{2}, e') \end{array}\right) \qquad \text{(Split/Merge)}$$

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

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

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

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

$$\Gamma; v \vdash (\, \pi.P\texttt{@}C\texttt{<}\bar{\pi}\texttt{>} * C \ \texttt{classof} \ \pi \,) \mathrel{-\!\!*} \pi.P\texttt{<}\bar{\pi}\texttt{>} \qquad \text{(Known Type)}$$

$$\Gamma; v \vdash \texttt{null}.\kappa\texttt{<}\bar{\pi}\texttt{>} \qquad \text{(Null Receiver)}$$

$$\Gamma; v \vdash \texttt{true} \qquad \text{(True)} \qquad\qquad \Gamma; v \vdash \texttt{false} \mathrel{-\!\!*} F \qquad \text{(False)}$$

$$\begin{array}{c}(\Gamma \vdash e, e' : T \ \wedge \ \Gamma, x : T \vdash F : \diamond) \\ \Rightarrow \Gamma; v \vdash (F[e/x] * e \texttt{ == } e') \mathrel{-\!\!*} F[e'/x]\end{array} \qquad \text{(Substitutivity)}$$

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

$$\Gamma; v \vdash \begin{array}{c}(\texttt{PointsTo}(e.f, \pi, e') \texttt{ \& } \texttt{PointsTo}(e.f, \pi', e'')) \\ \texttt{assures} \ \ e' \texttt{ == } e''\end{array} \qquad \text{(Unique Value)}$$

$$\begin{array}{c}(\Gamma \vdash e : T) \ \Rightarrow \ \Gamma; v \vdash (\texttt{ex} \ T \, \alpha)(e \texttt{ == } \alpha) \\ \Gamma; v \vdash (F \texttt{ \& } e) \mathrel{-\!\!*} (F * e)\end{array} \qquad \begin{array}{l}\text{(Well-typed)} \\ \text{(Copyable)}\end{array}$$

Table 1: Overview of axioms

Recall that we write "$F$ `assures` $G$" to abbreviate "$F$ `-* (`$F * G$`)`" (see Section 3.1.1). Then, the axiom (Well-typed) captures that all well-typed closed expressions represent a value (because built-in operations are total). And finally, (Copyable) expresses copyability of boolean expressions.

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

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

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

**Theorem 2** (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 is by induction on $(\Gamma; o; \bar{F} \vdash G)$'s proof tree. The pen and paper proof can be found in [37, §R]. □

### 3.2. Hoare Triples

In this section, we present Hoare rules to verify programs written in Section 2.1's language. Appendix B of Hurlin's PhD thesis [30] lists the complete collection of Hoare rules, presented in this and the next sections.

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

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

First, we present the rule for field writing . The rule's precondition[9] requires that the heap contains at least the object dereferenced and the field mentioned. In addition, it requires permission 1 to this object's field, i.e., write-permission. The rule's postcondition simply ensures that the heap has been updated with the value assigned. It should be noted that this rule is *small* [40]: it does not require anything more than a single `PointsTo` predicate. The (Frame) rule (discussed below) is used to build proofs in bigger contexts.

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

The rule for field reading requires a `PointsTo` predicate with *any* permission $\pi$:

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

The rule for creating new objects has precondition `true`, because we do not check for out of memory errors. After creating an object, all its fields are writable: the $\ell$`.init` predicate (formally defined in Appendix A.1) `*`-conjoins the predicates `PointsTo(`$\ell.f, 1, \mathsf{df}(T)$`)` for all fields $T\ f$ in $\ell$'s class:

[9] Where `PointsTo(`$u.f, 1, W$`)` abbreviates `(ex` $W\ w$`)(PointsTo(`$u.f, 1, w$`))`, as defined in Section 3.1.1.

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

The rule for method calls is verbose, but standard:

$$\frac{\begin{array}{c}\texttt{mtype}(m, t\texttt{<}\bar{\pi}\texttt{>}) = \texttt{<}\bar{T}\,\bar{\alpha}\texttt{> requires}\,G; \texttt{ensures (ex}\,U\,\alpha')\texttt{(}G'\texttt{)}; \\ U\ m\ \texttt{(}t\texttt{<}\bar{\pi}\texttt{>}\ \imath_0 \bar{W}\,\bar{\imath}\texttt{)} \\ \sigma = (u/\imath_0, \bar{\pi}'/\bar{\alpha}, \bar{w}/\bar{\imath}) \quad \Gamma \vdash u, \bar{\pi}', \bar{w} : t\texttt{<}\bar{\pi}\texttt{>}, \bar{T}[\sigma], \bar{W}[\sigma] \quad U[\sigma] <: \Gamma(\ell)\end{array}}{\Gamma; v \vdash \{u\ \texttt{!=null *}\ G[\sigma]\}\ell\texttt{=}u.m\texttt{(}\bar{w}\texttt{)}\{\texttt{(ex}\,U[\sigma]\ \ \alpha'\texttt{)(}\alpha'\ \texttt{==}\ \ell\ \texttt{*}\ G'[\sigma]\texttt{)}\}} \quad \text{(Call)}$$

Figure 1 lists the remaining standard rules, including rules for commands. Our judgment for *commands* combines typing and Hoare triples:

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

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

The last two rules in Figure 1 deserve some attention. The first is a rule for a specification-only `assert` statement, that will be formally defined on page 29. Intuitively, $\texttt{assert(}G\texttt{)}$ expresses that $G$ should hold at that point in the execution. It will be used to express a corollary about partial correctness of a verified program. The second rule 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.

Importantly, our system includes the (Frame) rule, which allows to reason locally about methods. To understand this rule, note that $\mathsf{fv}(F)$ is the set of free variables of $F$ and that we write $x \notin F$ to abbreviate $x \notin \mathsf{fv}(F)$. Furthermore, we write $\mathsf{writes}(hc)$ for the set of read-write variables $\ell$ that occur freely on the left-hand-side of an assignment in $hc$. (Frame)'s side condition on variables is standard [15, 21]. Bornat showed how to get rid of this side condition [41] by treating variable as resources.

### 3.3. Verified Interfaces and Classes

Before defining a judgment for verified programs, we need to define judgments for verified interfaces and classes. To do this, we first define method subtyping and predicate subtyping.

*Method Subtyping.* First, recall that *method types* are of the following form:

$$\texttt{<}\bar{T}\,\bar{\alpha}\texttt{> requires}\,F; \texttt{ensures}\,G; U\,m\texttt{(}V_0\,\imath_0; \bar{V}\,\bar{\imath}\texttt{)}$$

The self-parameter ($\imath_0$) is explicit, separated from the other formal parameters by a comma. Before presenting the method subtyping rule in full generality, we present its instance for method types without logical parameters:

$$\frac{\begin{array}{l}U, V_0, \bar{V}' <: U', V_0', \bar{V} \\ \Gamma, \imath_0 : V_0, \bar{\imath} : \bar{V}';\ \imath_0;\ \texttt{true} \vdash F'\ \texttt{-*}\ \texttt{(}F\ \texttt{*}\ \texttt{(fa}\,U\,\texttt{result)(}G\ \texttt{-*}\ G'\texttt{))}\end{array}}{\begin{array}{c}\Gamma \vdash \texttt{requires}\,F; \texttt{ensures}\,G; U\,m\texttt{(}V_0\,\imath_0, V\,\bar{\imath}\texttt{)} \\ <: \texttt{requires}\,F'; \texttt{ensures}\,G'; U'\,m\texttt{(}V_0'\,\imath_0, \bar{V}'\,\bar{\imath}\texttt{)}\end{array}}$$

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

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

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

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

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

$$\frac{\begin{array}{c}\Gamma; v \vdash \{F'\} hc \{G'\} \\ \Gamma; v; F \vdash F' \quad \Gamma; v; G' \vdash G\end{array}}{\Gamma; v \vdash \{F\} hc \{G\}} \text{ (Consequence)}$$

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

$$\frac{\Gamma \vdash w : \Gamma(\ell)}{\Gamma; v \vdash \{\mathtt{true}\} \ell = w \{\ell == w\}} \text{ (Var Set)}$$

$$\frac{\Gamma \vdash op(\bar{w}) : \Gamma(\ell)}{\Gamma; v \vdash \{\mathtt{true}\} \ell = op(\bar{w}) \{\ell == op(\bar{w})\}} \text{ (Op)}$$

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

$$\frac{\Gamma \vdash e, F : \mathtt{bool}, \diamond \quad \Gamma; v \vdash \{F\ \mathtt{\&}\ e\} c : \mathtt{void} \{F\}}{\Gamma; v \vdash \{F\} \mathtt{invariant}\ F;\ \mathtt{while}(e) \{c\} : \mathtt{void} \{F\ \mathtt{\&}\ \mathtt{!}e\}} \text{ (While)}$$

$$\frac{\Gamma; v; F \vdash G}{\Gamma; v \vdash \{F\} \mathtt{assert}(G) \{F\}} \text{ (Assert)}$$

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

Figure 1: Hoare triples

To understand this rule, we invite the reader to consider the following two derived rules (where types are elided):

$$\frac{\vdash F' \texttt{ -* } F \quad \vdash G \texttt{ -* } G'}{\vdash \texttt{requires } F\texttt{; ensures } G \texttt{ <: requires } F'\texttt{; ensures } G'}$$

$$\frac{}{\vdash \texttt{requires } F\texttt{; ensures } G \texttt{ <: requires } F \texttt{ * } H\texttt{; ensures } G \texttt{ * } H}$$

The first of these derived rules is standard behavioral subtyping, the second one abstracts separation logic's frame rule. In order to see that these two rules follow from the above rule, note that the following two formulas are tautologies (as can be easily proven by natural deduction):

$$\texttt{(}F' \texttt{ -* } F\texttt{) * } H \texttt{ -* } F' \texttt{ -* } F \texttt{ * } H \qquad F \texttt{ * } H \texttt{ -* } F \texttt{ * (fa } U\ x\texttt{)(}G \texttt{ -* } G \texttt{ * } H\texttt{)}$$

The general method subtyping rule also accounts for logical parameters:

$$\frac{\begin{array}{l}\Gamma, \imath_0 : V_0; \imath_0; \texttt{true} \vdash \texttt{(fa}\bar{T}'\ \bar{\alpha}\texttt{)(fa}\bar{V}'\ \bar{\imath}\texttt{)(}F' \texttt{ -* } \\ \qquad\qquad\qquad\qquad\qquad\qquad \texttt{(ex } \bar{W}\ \bar{\alpha}'\texttt{)(}F \texttt{ * (fa } U\ \texttt{result)(}G \texttt{ -* } G'\texttt{)))}\end{array}}{\begin{array}{l}\Gamma \vdash \texttt{<}T\ \bar{\alpha}, W\ \bar{\alpha}'\texttt{> requires } F\texttt{; ensures } G\texttt{; } U\ m\texttt{(}V_0\ \imath_0\texttt{; }V\ \bar{\imath}\texttt{)} \\ \qquad \texttt{<: <}\bar{T}'\ \bar{\alpha}\texttt{> requires } F'\texttt{; ensures } G'\texttt{; } U'\ m\texttt{(}V_0'\ \imath_0\texttt{; }\bar{V}'\ \bar{\imath}\texttt{)}\end{array}}$$

Note that the subtype may have more logical parameters than the supertype. For instance, we obtain the following derived rule:

$$\frac{}{\begin{array}{l}\vdash \texttt{<}T\ \alpha\texttt{> requires } F\texttt{; ensures } G \\ \qquad \texttt{<: requires (ex } T\ \alpha\texttt{)(}F\texttt{); ensures (ex } T\ \alpha\texttt{)(}G\texttt{)}\end{array}}$$

This derived rule is an abstraction of separation logic's auxiliary variable rule (Exists) (see page 24). It follows from the method subtyping rule by the following tautology:

$$\texttt{(ex } T\ \alpha\texttt{)(}F\texttt{) -* (ex } T\ \alpha\texttt{)(}F \texttt{ * (fa } U\ x\texttt{)(}G \texttt{ -* (ex } T\ \alpha\texttt{)(}G\texttt{)))}$$

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

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

*Class Axioms.* Recall that we use `axioms` to export useful relations between predicates to clients. For this to be sound, programmers have to prove axioms sound. We require that class axioms are proven sound with a restricted logical consequence judgment:

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

We disallow the application of class axioms for proving class axioms in order to avoid circularities. A class is sound if all its axioms are sound (the lookup function for axioms (axiom) is defined in Appendix A.1):

$$C\texttt{<}\bar{T}\,\bar{\alpha}\texttt{>}\ \mathsf{sound}$$
$$\text{iff}$$
$$\mathsf{axiom}(C\texttt{<}\bar{\alpha}\texttt{>}) = F \ \Rightarrow\ \bar{\alpha} : \bar{T}, \texttt{this} : C\texttt{<}\bar{\alpha}\texttt{>};\ \texttt{this};\ C\ \texttt{classof this} \vdash' F$$

*Class Extensions and Interface Implementations.* To define sanity conditions on classes and interfaces, we define some lookup functions. Let classes be of the form `class` $C\texttt{<}\bar{T}\ \bar{\alpha}\texttt{>}$ `ext` $U$ `impl` $\bar{V}$ `{`$fd$`*` $pd$`*` $ax$`*` $md$`*}` and interfaces of the form `interface` $I\texttt{<}\bar{T}\ \bar{\alpha}\texttt{>}$ `ext` $\bar{U}$ `{`$pt$`*` $ax$`*` $mt$`*}`. Then:

$$\begin{array}{rcl@{\qquad}rcl} \mathsf{methods}(C) & \triangleq & \mathsf{dom}(md\texttt{*}) & \mathsf{preds}(C) & \triangleq & \mathsf{dom}(pd\texttt{*}) \\ \mathsf{methods}(I) & \triangleq & \mathsf{dom}(mt\texttt{*}) & \mathsf{preds}(I) & \triangleq & \mathsf{dom}(pt\texttt{*}) \\ \mathsf{declared}(C) & \triangleq & \mathsf{dom}(fd\texttt{*}) & & & \end{array}$$

Now, we define sanity conditions on classes and interfaces. These conditions are later used to ensure that we only verify sane programs.

In the definitions below, we conceive the partial functions mtype and ptype (which are formally defined in Appendix A.1) as total functions that map elements outside their domains to the special element undef. Furthermore, we extend the subtyping relation: $<: = \{(T, U) \mid T <: U\} \cup \{(\mathsf{undef}, \mathsf{undef})\}$.

Judgment $C\texttt{<}\bar{T}\ \bar{\alpha}\texttt{>}$ ext $U$ expresses that: (1) class $C$ extends another class $U$, (2) class $C$ does not redeclare inherited fields, and (3) methods and predicates overridden in class $C$ are subtypes of the corresponding methods and predicates implemented in class $U$:

$$C\texttt{<}\bar{T}\ \bar{\alpha}\texttt{>}\ \mathsf{ext}\ U \triangleq \left\{ \begin{array}{l} U \text{ is a parameterized class} \\ f \in \mathsf{dom}(\mathsf{fld}(U)) \ \Rightarrow\ f \notin \mathsf{declared}(C) \\ \quad (\forall m, mt)(\mathsf{mtype}(m, U) = MT \ \Rightarrow \\ \qquad\qquad \bar{\alpha} : \bar{T} \vdash \mathsf{mtype}(m, C\texttt{<}\bar{\alpha}\texttt{>}) <: MT) \\ (\forall P, pt)(\mathsf{ptype}(P, U) = pt \ \Rightarrow\ \mathsf{ptype}(P, C\texttt{<}\bar{\alpha}\texttt{>}) <: pt) \end{array} \right.$$

Judgment $I\texttt{<}\bar{T}\ \bar{\alpha}\texttt{>}$ type-extends $U$ expresses that: (1) interface $I$ extends another interface $U$ and (2) methods and predicates overridden in interface $I$ are subtypes of the corresponding methods and predicates declared in $U$.

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

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

Judgment $C\texttt{<}\bar{T}\ \bar{\alpha}\texttt{>}$ impl $U$ expresses that: (1) class $C$ implements an interface $U$, (2) methods and predicates declared in interface $U$ are implemented in $C$, and

(3) methods and predicates implemented in $C$ are subtypes of the corresponding methods and predicates declared in $U$:

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

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

*Verified Interfaces and Classes.* In this paragraph, we define what are verified interfaces and classes. Later, when we verify a user-provided program, we will assume that the class table (i.e., a set of interfaces and classes) is verified.

**Well-formed Predicate Types, $\Gamma \vdash pt : \diamond$, and Method Types, $\Gamma \vdash mt : \diamond$:**

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

(Mth Type)
$$\frac{\Gamma, \bar{\alpha} : \bar{T}, \bar{\imath} : \bar{V} \vdash \bar{T}, F, U, \bar{V} : \diamond \quad \Gamma, \bar{\alpha} : \bar{T}, \bar{\imath} : \bar{V}, \texttt{result} : U \vdash G : \diamond}{\Gamma \vdash \texttt{<}\bar{T}\ \bar{\alpha}\texttt{>}\ \texttt{requires}\ F; \texttt{ensures}\ G;\ U\ m\texttt{(}\bar{V}\ \bar{\imath}\texttt{)} : \diamond}$$

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

(Ax)
$$\frac{\Gamma \vdash F : \diamond}{\Gamma \vdash \texttt{axiom}\ F : \diamond}$$

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

Below, we write $\mathsf{cfv}(c)$ for the set of variables that occur freely in an object creation command in $c$. Formally:

$$\mathsf{cfv}(c) = \{\alpha \in \mathsf{fv}(c) \mid \alpha \text{ occurs in an object creation command } \ell\texttt{ = new }C\texttt{<}\bar{\pi}\texttt{>}\ \}$$

Rule (Cls) below is the main judgment for verifying classes. Premises $C\texttt{<}\bar{T}\ \bar{\alpha}\texttt{>}\ \mathsf{ext}\ U$ and $C\texttt{<}\bar{T}\ \bar{\alpha}\texttt{>}\ \mathsf{impl}\ \bar{V}$ enforce class $C$ to be sane. Premise $C\texttt{<}\bar{T}\ \bar{\alpha}\texttt{>}\ \mathsf{sound}$ enforces $C$'s axioms to be sound. Premise $\bar{\alpha} : \bar{T}, \texttt{this} : C\texttt{<}\bar{\alpha}\texttt{>} \vdash fd\texttt{*}, ax\texttt{*}, md\texttt{*} : \diamond$ enforces $C$'s methods ($md\texttt{*}$) 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, $cl : \diamond$:**

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

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

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

### 3.4. Verified Programs

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, $st : \diamond$, that is preserved by the small-step rules of our operational semantics.

Our forcing relation $\models$ from Section 3.1.4 assumes formulas without logical variables: we deal with those by substitution, ranged over by $\sigma \in \mathsf{LogVar} \rightharpoonup \mathsf{SpecVal}$. We let $(\Gamma \vdash \sigma : \Gamma')$ whenever $\mathsf{dom}(\sigma) = \mathsf{dom}(\Gamma')$ and $(\Gamma[\sigma] \vdash \sigma(\alpha) : \Gamma'(\alpha)[\sigma])$ for all $\alpha$ in $\mathsf{dom}(\sigma)$.

Now, we extend the Hoare triple judgment to states:

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

The rule for states ensures that there exists a resource $\mathcal{R}$ 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". We have shown the following theorem:

**Theorem 3** (Preservation)**.** *If* $(ct : \diamond)$, $(st : \diamond)$ *and* $st \rightarrow_{ct} st'$, *then* $(st' : \diamond)$.

**Proof.** See [30, Chap. 6]. □

The proof of Theorem 3 means soundness of all rules presented so far. In particular, in this proof, we show that the Hoare rules from Section 3.2 are sound.

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 $c$ are sound (written $(ct, c) : \diamond$) iff ($ct : \diamond$ and $\texttt{null}; \emptyset \vdash$

$\{\texttt{true}\}c : \texttt{void}\{\texttt{true}\}$). 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. `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.

We can now state the first corollary (no `null` dereference) of the preservation theorem. A head command $hc$ is called a *null error* iff $hc = (\ell\,\texttt{=}\,\texttt{null}.f)$ or $hc = (\texttt{null}.f\,\texttt{=}\,v)$ or $hc = (\ell\,\texttt{=}\,\texttt{null}.m\texttt{<}\bar{\pi}\texttt{>}(\bar{v}))$ for some $\ell, f, v, m, \bar{\pi}, \bar{v}$.

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

**Proof.** See [30, Chap. 6]. □

To state the second corollary (partial correctness) 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:

$$\begin{array}{rcl} hc \in \mathsf{HeadCmd} & ::= & \ldots \mid sc \mid \ldots \\ sc \in \mathsf{SpecCmd} & ::= & \texttt{assert(}F\texttt{)} \end{array}$$

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

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

(Red No Op)

$$\ldots \quad \langle h,\, sc;\, c, s\rangle \rightarrow \langle h,\, c, s\rangle \quad \ldots$$

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

**Theorem 5** (Partial Correctness)**.**
*If* $(ct, c) : \diamond$ *and* $\mathsf{init}(c) \rightarrow_{ct}^{*} \langle h, \texttt{assert(}F\texttt{)};\, c,\, s\rangle$*, 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* $\sigma \in \mathsf{LogVar} \rightharpoonup \mathsf{SpecVal}$.

**Proof.** See [30, Chapter 6]. □

### 3.5. Example: Sequential mergesort

To show how the verification system works, we use it to verify a (sequential) implementation of mergesort. In the next section, when we add multithreading, we illustrate this by extending this example to parallel mergesort.

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.

```
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 2: Implementation of class `List`

Our example contains two classes: `List` and `MergeSort`, defined in Figure 2 and 3, respectively[10].

*Class* `List` This class contains three methods: method `append` adds a value to the tail of the list, method `concatenate(l,i)` concatenates the `i`-th first elements of list `l` to the receiver list, and method `get` returns the sub-tail of the receiver starting at the `i`-th element. It should be noted 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 `l`. In the first case, this means that the executing method should have permission to access the whole list. In the second case, however, it suffices to have access access to the inspected list up to a certain length.

[10]For simplicity of presentation, these classes are written using a more flexible language than Section 2's language. E.g. we allow reading of fields in conditionals and write chains of fields dereferencing.

To specify class `List`, we define the following two predicates.

```
class List extends Object{

 public pred state<nat n> = (n==0 -* True) *
        (n==1 -* [ex List l. PointsTo(next,1,l) * Perm(val,1)]) *
        (n>1  -* [ex List l. PointsTo(next,1,l) * Perm(val,1) *
                  l!=null * l.state<n-1>]);

 public pred state<nat n,List l> = (n==0 -* True) *
        (n==1 -* [PointsTo(next,1,l) * Perm(val,1)]) *
        (n>1  -* [ex List lb. PointsTo(next,1,lb) * Perm(val,1) *
                  lb!=null * lb.state<n-1,l>]);

}
```

Both predicates are `public`, because their definitions are necessary to prove class `MergeSort` correct.

Predicate `state<n>` gives access to the first `n` elements of the receiver list. Predicate `state<n,l>` has the same meaning as `state<n>`, but it also requires the successor of the `n`-th element to point to list `l`. Both predicates ensure that the receiver list is at least of length `n`, because of the test for non-nullness of the next element (`l!=null` and `lb!=null`). As a consequence, predicate `state<n,null>` represents a list of *exact length* `n`.

Equipped with these two predicates, we can specify class `List` as follows:

```
class List extends Object{

 requires init; ensures state@List<1,null>;
 List init(val v)

 requires state<i,null> * i>0; ensures state<i+1,null>;
 void append(int v)

 requires state<j,null> * j>0 * l.state<k> * k>=i;
 ensures  state<j+1,null> * l.state<k>;
 void concatenate(List l,int i)

 requires state<j> * j>=i * i>=0;
 ensures  state<i,result> * result.state<j-i>;
 List get(int i)

}
```

Method `init`'s postcondition refers 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. E.g. after calling $l$ `= new List` and $l$`.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 $l$`.state@List<1,null>` to $l$`.state<1,null>` (using axiom (Known Type)).

Because they are standard, we do not detail the proofs of the methods in class `List`.

*Class* `MergeSort` *This class (defined in Figure 3 on page 33)* implements the mergesort algorithm. The implementation is distributed over its `sort` method and its `merge` method. Class `MergeSort` has two fields: a pointer to the list so that it can be inspected, and an integer indicating how many nodes to inspect. 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. While method `sort` sorts the list *in place*, method `merge` allocates a new list `res` that is assigned to field `list`.

We prove that mergesort is memory safe 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 to mimic Java's lists.

To perform the proofs, we specify class `MergeSort` as shown below. 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()`.

```
class MergeSort<int length> extends Thread{

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

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

 requires state; ensures  result.state<length>;
 List sort()

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

In our proofs, we use a `split` axiom stating 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. This axiom is proved by induction over `n`:

```
(m1+m2==n * state<n>) *-* (ex List l.  state<m1,l> * l.state<m2>);
```

We have another axiom, named `forget-tail` that relates the two versions of predicate `state`:

```
state<n,l> -* state<n>
```

Axiom `forget-tail` allows - for example - to obtain the access ticket `state<1>` after a call to `init` (in combination with axiom (Known Type)).

```
class MergeSort extends Thread{
 List list; int num;
 void init(List l, int i){ list = l; num = i; }
 List sort(){
  if(num == 1){ list;}
  else{
  if(num == 2){
   if(list.val > list.next.val){
    int lval = list.val;
    list.val = list.next.val;
    list.next.val = lval;
   }
   list;
  }
  else{
     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 left = new MergeSort; left.init(list,lenleft);
     MergeSort right = new MergeSort;
     right.init(list.get(lenleft),lenright);
     List res = merge(left.sort(),lenleft,right.sort(),lenright);
     res;
  }
 }
 List merge(List ll, int lenleft, List rl, int lenright){
  List res;
  // This conditional avoids testing for nullity of res
  // inside the loop.
  if(ll.val <= rl.val){ res = new List; res.init(ll.val);
                        ll = ll.next; lenleft-=1; }
  else{ res = new List; res.init(rl.val,null);
        rl = rl.next; lenright-=1; }
  while(lenleft > 0 && lenright > 0){
   if(ll.val <= rl.val){ res.append(ll.val);
                         ll = ll.next; lenleft-=1; }
   else{ res.append(rl.val); rl = rl.next; lenright-=1; }
  }
  if(lenleft > 0) res.append(ll,lenleft);
  else { if(lenright > 0) res.append(rl,lenright); }
  res;
 }
}
```

Figure 3: Implementation of the sequential mergesort algorithm

## 4. Separation Logic for dynamic threads

In this section, we extend Section 2's language with multiple threads with fork and join primitives, à la Java. In addition, the assertion language and verification rules are extended to deal with these primitives. The rules allow to transfer permissions between threads upon thread creation and termination. We show that the resulting program logic is still sound, and we illustrate its use on a concurrent implementation of the merge sort algorithm.

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

### 4.1. A Java-like Language with fork/join

First, we extend the syntax and semantics of Section 2.1's language with fork and join primitives.

*Syntax.* 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 }
}
```

The methods `fork` and `join` are not implemented. As in Java, they are assumed to be implemented natively and their behavior is specified in 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.
- $o$.`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` method together with super calls as future work, and stick to our setting for simplicity.

*Runtime Structures.* In Section 2.2, our operational semantics $\rightarrow_{ct}$ was 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 "$s$ $\mathsf{in}$ $c$" for threads $t = (s, c)$, and "$o_1$ $\mathsf{is}$ $t_1$ $\mid \cdots \mid$ $o_n$ $\mathsf{is}$ $t_n$" for thread pools $ts = \{o_1 \mapsto t_1, \ldots, o_n \mapsto t_n\}$:

$$\begin{array}{rcl@{\quad}c@{\quad}l} t \in \mathsf{Thread} & = & \mathsf{Stack} \times \mathsf{Cmd} & ::= & s \ \mathsf{in}\ c \\ ts \in \mathsf{ThreadPool} & = & \mathsf{ObjId} \rightharpoonup \mathsf{Thread} & ::= & o_1 \ \mathsf{is}\ t_1 \mid \cdots \mid o_n \ \mathsf{is}\ t_n \\ st \in \mathsf{State} & = & \mathsf{Heap} \times \mathsf{ThreadPool} & & \end{array}$$

*Initialization.* We modify Section 2.2's definition of the initial state of a program. 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 $\emptyset$):

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

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) $\quad m \notin \{\texttt{fork}, \texttt{join}\}$

$$h(o)_1 = C\texttt{<}\bar{\pi}\texttt{>} \quad \mathsf{mbody}(m, C\texttt{<}\bar{\pi}\texttt{>}) = (\imath_0; \bar{\imath}).c_m \quad c' = c_m[o/\imath_0, \bar{v}/\bar{\imath}]$$
$$\langle h,\ ts \mid p\ \mathsf{is}\ (s\ \mathsf{in}\ \ell = o.m(\bar{v});\ c)\rangle \rightarrow \langle h,\ ts \mid p\ \mathsf{is}\ (s\ \mathsf{in}\ \ell \leftarrow c';\ c)\rangle$$

(Red Fork)
$$h(o)_1 = C\texttt{<}\bar{\pi}\texttt{>} \quad o \notin (\mathsf{dom}(ts) \cup \{p\})$$
$$\mathsf{mbody}(\texttt{run}, C\texttt{<}\bar{\pi}\texttt{>}) = (\imath).c_r \quad c_o = c_r[o/\imath]$$
$$\langle h,\ ts \mid p\ \mathsf{is}\ (s\ \mathsf{in}\ \ell = o.\texttt{fork()};\ c)\rangle \rightarrow \langle h,\ ts \mid p\ \mathsf{is}\ (s\ \mathsf{in}\ \ell = \texttt{null};\ c) \mid o\ \mathsf{is}\ (\emptyset\ \mathsf{in}\ c_o)\rangle$$

(Red Join)
$$\langle h,\ ts \mid p\ \mathsf{is}\ (s\ \mathsf{in}\ \ell = o.\texttt{join()};\ c) \mid o\ \mathsf{is}\ (s'\ \mathsf{in}\ v)\rangle \rightarrow$$
$$\langle h,\ ts \mid p\ \mathsf{is}\ (s\ \mathsf{in}\ \ell = \texttt{null};\ c) \mid o\ \mathsf{is}\ (s'\ \mathsf{in}\ v)\rangle$$

...

*Remarks.* In (Red Fork), a new thread $o$ is forked. Thread $o$'s state consists of an empty stack $\emptyset$ and command $c_0$. 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 checks whether thread $o$ is terminated because its command consists of a single value $v$.

### 4.2. Separation Logic for fork/join

In this section, we extend our assertion language to deal with `fork` and `join` primitives. We introduce (1) a `Join` predicate that controls how threads access postconditions of terminated threads and (2) `groups` which are a restricted class of predicates. Groups are needed to allow multiple threads to join a terminated thread.

#### *4.2.1. The* `Join` *predicate*

To model `join`'s behavior, we add a new formula to the assertion language defined in Section 3.1. This formula will be used (see Section 4.3) to govern

exchange of permissions from terminated threads to alive threads:

$$F \ ::= \ \ldots \mid \texttt{Join}(e, \pi) \mid \ldots$$

The intuitive meaning of $\texttt{Join}(e, \pi)$ is as follows: it allows to pick up fraction $\pi$ of thread $e$'s postcondition after $e$ terminated. As a specific case, if $\pi$ is 1, the thread in which the `Join` predicate appears can pick up thread $e$'s entire postcondition when $e$ terminates.

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:

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

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

*Resources.* To express the semantics of the `Join` predicate, we need to change our definition of models (resources). Recall that, in Section 3.1.2, resources were pairs of a heap and a permission table of type $\mathsf{ObjId} \times \mathsf{FieldId} \to [0, 1]$. Now, we modify permission tables so that they have type $\mathsf{ObjId} \times (\mathsf{FieldId} \times \{\texttt{join}\}) \to [0, 1]$. The additional element in the domain of permission tables keeps track of how much a thread can pick up of another thread's postcondition. In other words, in the previous chapter, the meaning of resources was to regulate access (write access, readonly access, or no access) to field of objects; now resources still have this meaning, *and* they regulate access of threads to other threads's postconditions. Obviously, we forbid `join` to be a valid field identifier to avoid confusion between `join`'s special meaning and programmer-declared fields.

In addition, we add an additional element to resources; they become triples of a heap, a permission table, and a *join table* $\mathcal{J} \in \mathsf{ObjId} \to [0, 1]$. 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:

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

Because $\#$ is equality, join tables are "global": in the preservation proof, all resources will have the same join table[11].

Now, we require resources to satisfy these axioms (in addition to the axioms of Section 3.1.2):

(a) For all $o \notin \mathsf{dom}(h)$ and all $f$ (including `join`), $\mathcal{P}(o, f) = 0$ and $\mathcal{J}(o) = 1$.

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

(b) $\lambda\, o\, .\, \mathcal{P}(o, \texttt{join}) \leq \mathcal{J}$.

Axiom (a) ensures that all objects that are not yet allocated have minimal permissions. This is needed to prove soundness of the verification rule for allocating new objects. Axiom (b) is a technical condition needed to prove soundness of the verification rule for joining threads.

As usual, we define a projection operator:

$$(h, \mathcal{P}, \mathcal{J})_{\mathsf{join}} \triangleq \mathcal{J}$$

*Predicate Environments.* For technical reasons, we need to update the definition of predicate environments given in Section 3.1.3 (see page 16). Specifically, we need predicate environments to satisfy the following additional axiom:

(b) If $(\bar{\pi}, (h, \mathcal{P}, \mathcal{J}), r, \bar{\pi}') \in \mathsf{Dom}(\kappa)$, $o \in \mathsf{dom}(h)$, $\mathcal{P}(o, \texttt{join}) \leq x \leq \mathcal{J}(o)$, and $\mathcal{J}' = \mathcal{J}[o \mapsto x]$, then:

$$\mathcal{E}(\kappa)(\bar{\pi}, (h, \mathcal{P}, \mathcal{J}), r, \bar{\pi}') \leq \mathcal{E}(\kappa)(\bar{\pi}, (h, \mathcal{P}, \mathcal{J}'), r, \bar{\pi}')$$

Axiom (b) is used to update the global join table, because, when a thread is joined, its corresponding entry drops in all join tables.

*Semantics.* The semantics of the `Join` predicate is as follows:

$$\Gamma \vdash (h, \mathcal{P}, \mathcal{J}); s \quad \models \quad \texttt{Join}(e, \pi) \text{ iff } [\![e]\!]^h_s = o \text{ and } [\![\pi]\!] \leq \mathcal{P}(o, \texttt{join})$$

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

$$\Gamma; v \vdash \texttt{Join}(e, \pi) \mathbin{\texttt{*-*}} (\texttt{Join}(e, \tfrac{\pi}{2}) \mathbin{\texttt{*}} \texttt{Join}(e, \tfrac{\pi}{2})) \qquad \text{(Split/Merge Join)}$$

*4.2.2. Groups*

In order to express that multiple threads can join a terminated thread, we introduce the notion of *groups*. Groups are special predicates, denoted with keyword `group` that satisfy an additional split/merge axiom (whereas many predicates do not satisfy this axiom, see [30, 4.6.2]). Formally, `group` desugars to a `pred`icate and an `axiom`:

$$\texttt{group}\ P\texttt{<}\bar{T}\ \bar{x}\texttt{>} = F \quad \triangleq \quad \begin{array}{l} \texttt{pred}\ P\texttt{<}\bar{T}\ \bar{x}\texttt{>} = F\texttt{;} \\ \texttt{axiom}\ P\texttt{<}\bar{x}\texttt{>} \mathbin{\texttt{*-*}} (P\texttt{<}\texttt{split}(\bar{T}, \bar{x})\texttt{>} \mathbin{\texttt{*}} P\texttt{<}\texttt{split}(\bar{T}, \bar{x})\texttt{>}) \end{array}$$

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

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

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

### 4.3. Contracts for fork and join

Next, we discuss how the verification logic for the sequential language, presented in Section 3.2 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 (in contrast to the operational semantics). 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 threads.

*Class* `Thread`. In Section 4.1, 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 similar to `run`'s precondition and (2) `run`'s postcondition includes predicate `postJoin<1>` while `join`'s postcondition is `postJoin<p>`. Point (1) models that when a thread is forked, its `run` method is executed: part of the parent thread's state is transferred to the forked thread. Point (2) models that `join` returns after `run` terminates. Further, (2) represents that threads joining a thread might 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 a 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 threads joining $o$ do not pick up more than thread $o$'s postcondition.

Notice that soundness of the approach required to define `postJoin` as a group. Intuitively, this 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

a `final` modifier). 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>`). Fixing `run`'s contract in class `Thread` means that programmers have to specify `run` by adapting the predicates `preFork` and `postJoin`. In our examples, this proved to be convenient, however we have not investigated consequences of this choice on more intricate examples.

Since `run`'s contract is fixed, `run`'s contract cannot be parameterized by logical parameters. One could consider that this reduces expressiveness. But this is wrong, 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 `Executor`s [42]).

*Alternative Solutions.* Alternatively, we could allow arbitrary contracts for `run`, as we did in our earlier AMAST paper [28]. This solution, however, has the disadvantage that we need to introduce a new derived form for formulas (called *scalar multiplication*) at the level of method's contracts. With this paper's solution, we can "hide" scalar multiplication "under the carpet" (see Section 4.4), i.e., we avoid scalar multiplication to spread in method contracts and in proofs of programs (even if we need it to *prove* our verification system sound). An additional advantage of this approach is that it does not break subtyping. In our earlier work, to start a thread, one had to know the static type of the thread considered one level (in the class hierarchy) below class `Thread`.

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 [43]).

### 4.4. Verified Programs

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

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

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 a resource $\mathcal{R}$ 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 3.4). The new state rule looks as follows.

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

In addition to 3.4's state rule, the rule for individual threads 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 $\sum_{i=1}^{n} bit_i \cdot \frac{1}{2^i}$:

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

Intuitively, we use symbolic binary fractions to speak about finite formulas of the form $r.P\texttt{<}1\texttt{>} * r.P\texttt{<}\frac{1}{2}\texttt{>} * r.P\texttt{<}\frac{1}{8}\texttt{>} * \ldots$. Formally, we define the scalar multiplication $fr \cdot r.P\texttt{<}\pi\texttt{>}$ as follows:

$$\begin{aligned}
\mathsf{all} \cdot r.P\texttt{<}\pi\texttt{>} &= r.P\texttt{<}\pi\texttt{>} \\
\mathsf{fr}() \cdot r.P\texttt{<}\pi\texttt{>} &= \texttt{true} \\
\mathsf{fr}(1) \cdot r.P\texttt{<}\pi\texttt{>} &= r.P\texttt{<split}(\pi)\texttt{>} \\
\mathsf{fr}(0, bits) \cdot r.P\texttt{<}\pi\texttt{>} &= \mathsf{fr}(bits) \cdot r.P\texttt{<split}(\pi)\texttt{>} \\
\mathsf{fr}(1, bits) \cdot r.P\texttt{<}\pi\texttt{>} &= r.P\texttt{<split}(\pi)\texttt{>} * \mathsf{fr}(bits) \cdot r.P\texttt{<split}(\pi)\texttt{>}
\end{aligned}$$

For instance, $\mathsf{fr}(1,0,1) \cdot r.P\texttt{<}1\texttt{>} \mathbin{*\!\!-\!\!*} (r.P\texttt{<}\frac{1}{2}\texttt{>} * r.P\texttt{<}\frac{1}{8}\texttt{>})$. The map $[\![\cdot]\!] : \mathsf{BinFrac} \to \mathbb{Q}$ interprets symbolic binary fractions as concrete rationals:

$$[\![\mathsf{all}]\!] \triangleq 1 \quad [\![\mathsf{fr}()]\!] \triangleq 0 \quad [\![\mathsf{fr}(1)]\!] \triangleq \tfrac{1}{2}$$

$$[\![\mathsf{fr}(0, bits)]\!] \triangleq \tfrac{1}{2}[\![\mathsf{fr}(bits)]\!] \quad [\![\mathsf{fr}(1, bits)]\!] \triangleq \tfrac{1}{2} + \tfrac{1}{2}[\![\mathsf{fr}(bits)]\!]$$

Now, the rule for individual threads is as follows:

$$\frac{\begin{array}{c}\mathcal{R}_{\mathsf{join}}(o) \leq [\![fr]\!] \quad \Gamma \vdash \sigma : \Gamma' \quad \Gamma, \Gamma' \vdash s : \diamond \quad \mathsf{cfv}(c) \cap \mathsf{dom}(\Gamma') = \emptyset \\ \Gamma[\sigma] \vdash \mathcal{E}; \mathcal{R}; s \models F[\sigma] \quad \Gamma, \Gamma'; r \vdash \{F\}c : \texttt{void}\{\ fr \cdot o\texttt{.postJoin<1>}\ \}\end{array}}{\mathcal{R} \vdash o\ \mathsf{is}\ (s\ \mathsf{in}\ c) : \diamond} \quad \text{(Thread)}$$

In rule (Thread), $fr$ should be bigger than the thread considered's entry in the global join table (condition $\mathcal{R}_{\mathsf{join}}(o) \leq [\![fr]\!]$). This forces joining threads to take back a part of a terminated thread's postcondition which is smaller than the terminated thread's "remaining" postcondition. This comes from the semantics of the `Join` predicate and the semantics of join tables: $\Gamma \vdash (h, \mathcal{P}, \mathcal{J}); s \models \texttt{Join(}e, \pi\texttt{)}$ holds iff $[\![e]\!]^h_s = o$ and $[\![\pi]\!] \leq \mathcal{P}(o, \texttt{join})$. Moreover, by $\mathcal{P} \leq \mathcal{J}$ (see axiom (b) on page 37), we have that $\mathcal{P}(o, \texttt{join}) \leq \mathcal{J}(o)$.

As in Section 3.4, we have shown that the preservation theorem (Theorem 3) holds, and we have shown that verified programs satisfy the following properties: null error freeness and partial correctness.

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 $o, f, v, v', \ell$.

**Theorem 6** (Verified Programs are Data Race Free)**.** *If* $(ct, c) : \diamond$ *and* $\mathsf{init}(c) \rightarrow_{ct}^{*} \langle h,\ ts \mid o_1 \ \mathsf{is}\ (s_1 \ \mathsf{in}\ hc_1; c_1) \mid o_2 \ \mathsf{is}\ (s_2 \ \mathsf{in}\ hc_2; c_2)\rangle$*, then* $(hc_1, hc_2)$ *is not a data race.*

**Proof.** See [30, Chapter 6]. □

### 4.5. Example of reasoning: parallel mergesort

To show how the verification system works in a multithreaded setting, we use it to verify a parallel implementation of mergesort. In the previous chapter, we verified a sequential mergesort algorithm, so we concentrate on the changes in specification and verification because of the use of multiple threads. The multithreaded algorithm is a perfect example of disjoint parallelism, because the different threads all modify the *same* list simultaneously. This is sound, because the threads mutate the list in different places.

Our example reuses class `List` from Section 3.5 which we do not repeat here.

The changes to class `MergeSort` are shown in Figure 4. Class `MergeSort` now extends class `Thread`. The implementation of the mergesort algorithm is distributed over its `run` methods and its `merge` method. As in Section 3.5, the class has two fields: a pointer to the list so that it can be inspected, and an integer indicating how many nodes to inspect. Again, method `run` distinguishes three cases: (i) if there is only one node to inspect, or (ii) if there are only two nodes to inspect, or (iii) if the list's length is greater than 2. Contrary to Section 3.5, however, in the third case, two new threads are created to sort the left and the right parts of the list. Then, the parent threads waits for the two new threads and `merge`s their results. To perform the proofs, we specify class `MergeSort` as in Figure 5.

Figure 6 shows part of the correctness proof of method `run`. This proof 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 3.5) to mimic this behavior 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 of this list.

## 5. Separation Logic for Reentrant Locks

In this section, we present verification rules for Java's reentrant locks. Together with `fork` and `join`, reentrant locks are a crucial feature of Java for multi-threaded programs. Generally speaking, locks serve to synchronize threads and to control exclusive access to resources that cannot be shared.

Reentrant locks can be acquired more than once by the same thread. This differs from Posix threads that block if they acquire a lock twice. On one hand, reentrant locks are a convenient tool for programmers because code does not need to check if a lock is already acquired before trying to acquire it. This is

```
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.start();
    MergeSort<lenright> right = new MergeSort;
    right.init(tail,lenright); right.start();
    left.join();
    right.join();
    merge(left,right);
   }
  }
 }
 // very similar to the sequential version
 void merge(MergeSort left, MergeSort right){ ... }
}
```

Figure 4: Implementation of parallel mergesort algorithm

```
class MergeSort<int length> extends Thread{
 pred preFork = PointsTo(list,1,l) * PointsTo(num,1,n) *
                l!=null * n >= 1 * n==length * l.state<length>;
 pred postJoin<perm p> = preFork;
 requires init * l.state<length> * 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,l) * l.state<length>;
 void merge(MergeSort<nl> left, MergeSort<nr> right) {...}
}
```

Figure 5: Specification of class `MergeSort` (parallel version)

particularly useful for developing easy to use libraries, as illustrated e.g., by the example in Section 5.6.1. On the other hand, reentrant locks require extra machinery in the verification system, because initial acquirements have to be distinguished from reentrant acquirements.

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 illustrate the approach on some examples.

### 5.1. Background on Separation Logic and Locks

Separation logic for programming language with locks as a concurrency primitive has been first explored by O'Hearn [15]. O'Hearn elegantly adapted an old idea from concurrent programs with shared variables [14]. Each lock is associated with a *resource invariant* which describes the part of the heap that the lock guards. When a lock is acquired, it lends its resource invariant to the 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:

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

While these rules are sound for single-entrant locks, they are unsound for reentrant locks, because they allow to "duplicate" a lock's resource invariant:

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

```
 { PointsTo(list,1,l) * PointsTo(num,1,n) * l!=null * n >= 1 *
   l.state<n> * n==length }
  (Let F be the abbreviation of PointsTo(list,1,l) * PointsTo(num,1,n))
 {  F * l!=null * n >= 1 * l.state<n> * 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,f> * f.state<lenright> * 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,f> * f.state<lenright> * lenleft+lenright==length }
 }
  (In both cases, we have:)
  { F * n > 2 * n==length *
    l.state<lenleft,f> * f.state<lenright> * lenleft+lenright==length }
  ("split" axiom from right to left)
  { F * n > 2 * n==length *
     l.state<n> * 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> * tail.state<n-lenleft> }
  (axiom forget-tail and arithmetic (n-lenleft==lenright))
  { F * G * l.state<lenleft> * tail.state<lenright> }
 MergeSort<lenleft> left = new MergeSort; left.init(list,lenleft);
  { F * G * tail.state<lenright> * left.preFork * Join(left,1) }
 left.start();
  { F * G * tail.state<lenright> * Join(left,1) }
 MergeSort<lenright> right = new MergeSort; right.init(tail,lenright);
 right.start();
  { 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> }
  (Close)
  { postJoin<1> }
```

Figure 6: Correctness proof of method `run` in class `MergeSort`

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 to transfer 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 [15, 44, 20], we also handle inheritance.

### 5.2. A Java-like Language with Reentrant Locks

First we modify the syntax and the semantics of Section 4.1's language to model reentrant locks.

*Syntax.* We extend the list of head commands defined in Section 2.1 as follows:

$$hc \in \mathsf{HeadCmd} \quad ::= \quad \ldots \mid v.\texttt{lock()} \mid v.\texttt{unlock()} \mid \ldots$$

We use `lock` and `unlock` primitives, because they are more expressive than Java's `synchronized` blocks. First, `synchronized` blocks forbid to lock and unlock an object in different methods. Second, `synchronized` blocks enforce proper nesting of `lock` and `unlock` commands. For example the behavior of `o.lock(); q.lock(); o.unlock(); q.unlock();` cannot be reproduced with `synchronized` blocks. Moreover, Java 5 provides lock *objects* that provide a behavior similar to our `lock` and `unlock` primitives.

Like class invariants must be initialized before method calls, resource invariants must be initialized before the associated locks can be acquired. In O'Hearn's simple concurrent language [15], 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.

We distinguish between *fresh* locks and *initialized* locks. A fresh lock does not yet guard its resource invariant: a fresh lock 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. We require programmers to explicitly change the state of locks (from fresh to initialized) with a `commit` command:

$$sc \in \mathsf{SpecCmd} \quad ::= \quad \ldots \mid \pi.\texttt{commit} \mid \ldots$$

Operationally, $\pi$.`commit` is a no-op; semantically it checks that $\pi$ is fresh and changes $\pi$'s state to initialized. For expressiveness purposes, `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. Another possibility would be to infer `commit` commands automatically.

Like in Java, we assume that class tables always contain the following class declaration:

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

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`. As usual [15], the resource invariant $o$.`inv` can be assumed when $o$'s lock is acquired non-reentrantly and must be established when $o$'s lock is released with its reentrancy level dropping to 0. Regarding the interaction with subclassing, there is nothing special about `inv`. It is treated just like any other abstract predicate.

The methods `wait` and `notify` do not have Java implementations, but are implemented natively. To model this, our operational semantics specifies their behavior explicitly (see $\rightarrow$'s definition on page 48). Intuitively, these methods behave as follows:

- If $o$.`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 $o$.`notify()` is called, one thread that is stopped (because this thread called $o$.`wait()` before) resumes and starts competing for $o$'s lock. When a resumed thread reacquires $o$'s lock, its previous reentrancy level is restored.

Since we can specify method contracts for `wait` and `notify`, we do not put them in our set of commands. Their specification will be given in Section 5.4. In contrast, `lock`, `unlock`, and `commit` are put in our set of commands, because the Hoare rules for these methods cannot be expressed with the syntax of contracts available to programmers: we need extra expressivity (see Section 5.4).

In addition to `wait` and `notify`, Java provides method `notifyAll` to notify all threads waiting on an object. We do not include `notifyAll` in our verification system, however it can be treated exactly like `notify`.

*Runtime Structures.* To represent locks in the operational semantics, we use a *lock table*. *Lock tables* map objects $o$ to either the symbol $\mathsf{free}$, or to the thread object that currently holds $o$'s lock and a number that counts how often it currently holds this lock:

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

Compared to Section 4.1, states are extended to include a heap, a lock table, and a thread pool:

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

*Initialization.* We modify Section 4.1's definition of the initial state of a program. Initially, the lock table of a program is empty (hence the second $\emptyset$):

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

We modify the *operational semantics* defined in Section 4.1 to deal with locks. Except that a lock table is added, most of the existing cases of the operational semantics are left untouched.

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

$hc\ ::=\ \ldots\ \mid\ o.\texttt{waiting}(n)\ \mid\ o.\texttt{resume}(n)\ \mid\ \ldots$
*Restriction:* These clauses must not occur in source programs.

Here are the intuitive semantics of these head commands:

- $o.\texttt{waiting}(n)$: If thread $p$'s head command is $o.\texttt{waiting}(n)$, then $p$ is waiting to be notified to resume competition for $o$'s lock at reentrancy level $n$.
- $o.\texttt{resume}(n)$: If thread $p$'s head command is $o.\texttt{resume}(n)$, 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), and (Red Resume).

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

$\ldots$

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

(Red Call) $\quad m \notin \{\texttt{fork}, \texttt{join}, \texttt{wait}, \texttt{notify}\}$
$h(o)_1 = C\texttt{<}\bar{\pi}\texttt{>}\quad \mathsf{mbody}(m, C\texttt{<}\bar{\pi}\texttt{>}) = (\imath_0, \bar{\imath}).c_m\quad c' = c_m[o/\imath_0, \bar{v}/\bar{\imath}]$
$\langle h,\ l\ ,\ ts \mid p\ \mathsf{is}\ (s\ \mathsf{in}\ \ell = o.m(\bar{v});\ c)\rangle \rightarrow \langle h,\ l\ ,\ ts \mid p\ \mathsf{is}\ (s\ \mathsf{in}\ \ell \leftarrow c';\ c)\rangle$

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

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

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

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

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

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

...

*Remarks.*

- 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 ($l(o) = (n, p)$).
- Similarly, rule (Red Unlock) distinguishes two cases: (1) lock $o$'s reentrancy level decreases but $o$ remains acquired ($l(o) = (n, p)$ and $n > 1$) and (2) lock $o$ is released ($l(o) = (1, p)$).
- 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) and (Red Skip Notify) fire only if the thread considered previously acquired `notify`'s receiver. The rule (Red Notify) fires if there is a thread waiting on `notify`'s receiver. In this case, the waiting thread is resumed. If there is no thread waiting on `notify`'s receiver, rule (Red Skip Notify) fires. In this case, the call to `notify` has no effect on other threads.
- In Java, if $o$.`wait()` and $o$.`notify()` 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 5.4, we will give contracts for `wait` and `notify` that ensure verified programs are never stuck when calling `wait` or `notify`. In another words, verified programs would never throw an `IllegalMonitorState` exception in Java's semantics.
- Rule (Red Resume) resumes a thread that previously waited on some lock. The thread's reentrancy level is restored.

### 5.3. Separation Logic for Reentrant Locks

In this section, we describe the new formulas that we add to the specification language of Section 4.2.

In separation logic for single-entrant locks [15], locks can be acquired unconditionally. For reentrant locks, on the other hand, it seems unavoidable that the proof rule for acquiring a lock distinguishes between initial acquires and reacquires. This is needed because it is quite obviously unsound to simply assume the resource invariant after a re-acquire. Thus, a proof system for reentrant locks must keep track of the locks that the current thread holds. To this end, we enrich our specification language:

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

Here is the informal semantics of the new specification values and formulas:

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

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

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

*Initializing Resource Invariants.* As explained before, resource invariants must be initialized before the associated locks can be acquired. We use the specification command `commit` to indicate where a lock changes from the fresh state to the initialized state. Because we do not tie initialization to a specific program point (such as the end of constructors), we also have to keep track of the state of locks in our verification system. To this end, we introduce two more formulas:

$F \in$ Formula ::= ... | $e$`.fresh` | $e$`.initialized` | ...
*Restriction:* $e$`.initialized` must not occur in negative positions.

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

Because $e$`.initialized` is copyable, `initialized` formulas can "spread" to all threads, allowing all threads to try to acquire locks (in Section 5.4, we will see that `initialized` appears in the precondition of the Hoare rules for (initial) lock acquirement).

*Types.* We add a type to represent locksets:

$$T ::= \ldots \mid \texttt{lockset} \mid \ldots$$

It is convenient to allow using objects as singleton locksets (rather than introducing explicit syntax for converting from objects to singleton locksets). Hence, we postulate `Object` <: `lockset`.

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:** $\pi \simeq \pi$

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

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

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

*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):

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

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

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

We require resources to satisfy the following axioms (in addition to Section 4.2's axioms):

(a) $\mathcal{F} \cap \mathcal{I} = \emptyset$.
(b) If $o \in \mathcal{L}(p)$ then $o \in \mathcal{I}$.

Axiom (a) ensures that our interpretation of fresh sets and initialized sets makes sense: an object can never be both fresh *and* initialized. Axiom (b) ensures that locked objects are initialized.

As usual, we define projection operators:

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

*Predicate Environments.* In addition to Section 4.2's axioms, we require predicate environments to satisfy the following axiom:

(a) If $(\bar{\pi}, (h, \mathcal{P}, \mathcal{J}, \mathcal{L}, \mathcal{F}, \mathcal{I}), r, \bar{\pi}') \in \mathsf{Dom}(\kappa)$,
then $\mathcal{E}(\kappa)(\bar{\pi}, (h, \mathcal{P}, \mathcal{J}, \mathcal{L}, \mathcal{F}, \mathcal{I}), r, \bar{\pi}') \leq \mathcal{E}(\kappa)(\bar{\pi}, (h, \mathcal{P}, \mathcal{J}, \mathcal{L}, \mathcal{F}, \mathcal{I} \cup \{o\}), r, \bar{\pi}')$

This is a technical condition used to update the global initialized set when an object (here $o$) is committed.

*Semantics of Values.* Before defining the semantics of formulas, we need to extend the semantics of values to locksets. Recall that $\mathsf{SemVal}$ is the set of semantic values (defined in Section 3.1.4). Formally, $\mathsf{SemVal} = \{\texttt{null}\} \cup \mathsf{ObjId} \cup \mathsf{Int} \cup \mathsf{Bool} \cup (0, 1]$. We extend this set to include semantic domains for locksets. The resulting set of semantic values is defined as follows:

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

$$\Gamma; v \vdash \texttt{!(nil contains } e\texttt{)} \qquad \text{(Member Nil)}$$

$$\Gamma; v \vdash \texttt{(}\pi \cdot \pi'\texttt{) contains } e \texttt{ *-* (}e \texttt{ == } \pi \texttt{ | } \pi' \texttt{ contains } e\texttt{)} \qquad \text{(Member Rec)}$$

$$\pi \simeq \pi' \;\Rightarrow\; \Gamma; v \vdash \pi \texttt{ == } \pi' \qquad \text{(Eq Bag)}$$

$$\Gamma; v \vdash \pi \texttt{ == } \pi \qquad \text{(Eq Refl)}$$

$$\Gamma; v \vdash \pi \texttt{ == } \pi' \;\Rightarrow\; \Gamma; v \vdash \pi' \texttt{ == } \pi \qquad \text{(Eq Sym)}$$

$$\Gamma; v \vdash \pi \texttt{ == } \pi' \texttt{ \& } \pi' \texttt{ == } \pi'' \;\Rightarrow\; \Gamma; v \vdash \pi \texttt{ == } \pi'' \qquad \text{(Eq Trans)}$$

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

Table 2: Axioms to reasons about bags and copyable formulas

where $\equiv$ is the least equivalence relation on $\mathsf{SemVal}$ 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 $\mathsf{WellTypClSpecVal}$ be the set of well-typed, closed specification values[12]:

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

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

$$[\![.]\!] : \mathsf{WellTypClSpecVal} \to \mathsf{SemVal} \qquad [\![\texttt{nil}]\!] \triangleq [\,] \qquad [\![\pi \cdot \pi']\!] \triangleq [\![\pi]\!] \sqcup [\![\pi']\!]$$

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

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

These clauses are self-explanatory, except perhaps the existential quantification in the clause for `Lockset(`$\pi$`)`. Intuitively, this clause says that there exists a thread identifier $o$ in $\mathcal{L}$'s domain such that $\pi$ denotes the current lockset associated with $o$.

*Axioms.* Table 2 lists the new axioms. These axiomatize bag membership ((Member Nil) and (Member Rec)); bag equality ((Eq Bag)); equality between specification values ((Eq Refl), (Eq Sym), and (Eq Trans)); and copyability. Axiom (Copyable) updates Section 3.1.5's (Copyable) axiom about copyability of formulas.

[12]Recall that typing rules are defined in Appendix A.3.

### 5.4. Hoare Triples

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

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

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

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 gets around explicit reasoning about the absence of aliasing. Unfortunately, we cannot entirely avoid this kind of reasoning when establishing the precondition for the rule (Lock) below, which requires that the lock is *not* already held by the current thread.

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

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

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

The rule (Commit) ensures that monitor invariants cannot mention `Lockset` predicates. This is important because `Lockset` predicates are interpreted w.r.t. to the current thread: having `Lockset` predicates inside monitor invariants does not make sense. The fact that monitor invariants cannot mention `Lockset` predicates is enforced by (Commit)'s precondition: because it mentions both a `Lockset` predicate and the lock's monitor invariant `inv`, `inv` cannot include a `Lockset` predicate. This follows from the semantics of the `Lockset` predicate and the semantics of the `*` operator: two `Lockset` predicates cannot be `*`-conjoined. Hence, if `inv` includes a `Lockset` predicate, (Commit)'s precondition cannot be established.

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:

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

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

The rule (Lock) applies when lock $u$ is acquired non-reentrantly, as expressed by the precondition $\texttt{Lockset}(\pi) * \texttt{!}(\pi \texttt{ contains } u)$. The precondition $u.\texttt{initialized}$ 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 $u.\texttt{inv}$ (without `@` selector). In proofs, the "visible" resource invariant is opened at $u$'s static type using axioms (Open/Close) and (Dynamic Type) (see page 21).

Proving (Lock)'s precondition requires reasoning about aliases because one has to prove $\texttt{!}(\pi \texttt{ contains } u)$. In practice, this assertion is proven by showing that $u$ is different from all elements of lockset $\pi$. 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 5.6, the lock coupling example illustrates 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 $u.\texttt{initialized}$, because this follows from $\texttt{Lockset}(u \cdot \pi)$ (locksets contain only initialized values).

To provide more useful feedback to programmers, we present a derived rule of (Re-Lock) that could be used in program checkers instead of (Re-Lock).

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

```
requires Initialized * Lockset(S); ensures ...;
void syncCallToMth(){
  lock();
    mth(); // requires inv
  unlock();
}
```

Figure 7: Example showing (Re-Lock-Accurate)'s usefulness

Figure 7 illustrates why this rule provides more accurate feedback. Consider method `syncCallToMth`. To verify this method, a possible strategy for

a program checker is to make a case distinction on `(!S contains this | S contains this)` when reaching `lock()`. When verifying the the case where `S contains this` holds (i.e., `this` is acquired reentrantly), a program checker that uses (Re-Lock-Accurate) would fail at the call to `lock()`, because it would require $u$.`inv` when `lock()` is called. In contrast, a program checker that uses (Re-Lock) would fail at the call to `mth()`, because it does not fulfill its precondition. Thus, since the problem is actually due to the usage of locks, rule (Re-Lock-Accurate) provides more accurate feedback. In other words, upon lock reentrance, one expects this lock's resource invariant to hold, and this is what rule (Re-Lock-Accurate) enforces. Thus, rule (Re-Lock) is appropriate for the theory but rule (Re-Lock-Accurate) is more convenient in practice.

Next, we present the two rules for unlocking:

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

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

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 disequalities in (New)'s postcondition. Several approaches for this come to mind. First, one could drop the disequalities in (New)'s postcondition, and rely on (Commit)'s postcondition `!(`$\pi'$ `contains` $\pi$`)` to establish (Lock)'s precondition. While this would be sound, in general it is too weak, as we are unable to lock $\pi$ if we first lock some other object $x$ (because from `!(`$\pi'$ `contains` $\pi$`)` we cannot derive `!(`$x \cdot \pi'$ `contains` $\pi$`)` unless we know $\pi$ `!=` $x$). Second, the `Lockset` predicate could be abandoned altogether, using a predicate $\pi$.`Held(`$n$`)` instead, that specifies that the current thread holds lock $\pi$ with reentrancy level $n$. In particular, $\pi$.`Held(0)` means that the current thread does not hold $\pi$'s lock at all. We could reformulate the rules for locking and unlocking using the `Held`-predicate, and introduce $\ell$.`Held(0)` as the postcondition of (New), replacing the disequalities. However, this approach does not work, because it grants only the object creator permission to lock the created object! While it is possible that a clever program logic could somehow introduce $\pi$.`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.

*Wait and notify.* Recall that in Section 5.2, we added methods `wait` and `notify` in class `Object` without specifying their contracts. Now we specify those 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();
}
```

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

*Auxiliary Syntax.* Recall that in Section 5.2, we added two new head commands `waiting` and `resume` to represent waiting states. The Hoare rules for these commands are as follows:

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

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

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

### 5.5. Verified Programs

In this section, we show how to modify class `Thread` and top level rules of Section 4's verification system to handle reentrant locks.

*The* `Thread` *class.* To handle reentrant locks, we modify class `Thread`'s method contracts as shown in Figure 8. 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` differs from the newly created thread, while a thread calling `join` differs from 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:

```
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 8: Class `Thread`

*Top Level Rules.* We need to update Section 4.4's rules for runtime states to account for reentrant locks.

There are two changes to rule (Thread): (1) premise $\mathsf{dom}(\mathcal{R}_{\mathsf{lock}}) = \{o\}$ is added to ensure that a thread's resource 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)

$$\frac{\begin{array}{c}\mathcal{R}_{\mathsf{join}}(o) \leq [\![fr]\!] \quad \Gamma \vdash \sigma : \Gamma' \quad \Gamma, \Gamma' \vdash s : \diamond \\ \mathsf{cfv}(c) \cap \mathsf{dom}(\Gamma') = \emptyset \quad \mathsf{dom}(\mathcal{R}_{\mathsf{lock}}) = \{o\} \quad \Gamma[\sigma] \vdash \mathcal{E}; \mathcal{R}; s \models F[\sigma] \\ \Gamma, \Gamma'; r \vdash \{F\}c : \texttt{void}\{\ \texttt{(ex lockset}\,S\texttt{)(Lockset(}S\texttt{))}\ * fr \cdot o.\texttt{postJoin<1>}\}\end{array}}{\mathcal{R} \vdash o\ \mathsf{is}\ (s\ \mathsf{in}\ c) : \diamond}$$

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

$$\mathsf{ready}(\mathcal{R}) \triangleq \mathcal{R}_{\mathsf{init}} \setminus \{\, o \mid (\exists p)(o \in \mathcal{L}(p)) \,\}$$

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

In $\mathsf{conc}$'s definition, we let $\mathcal{L}(p)(o)$ stand for the multiplicity of $o$ in $\mathcal{L}(p)$. Note that $\mathsf{conc}$ is well-defined, by axiom (b) for resources (see page 50). The new rule for states ensures that there exists a resource $\mathcal{R}$ to satisfy the thread pool $ts$ and a resource $\mathcal{R}'$ to satisfy the resource invariants of the locks that are ready to be acquired. In addition, function $\mathsf{conc}$ relates the program's lock table to the top level resource's abstract lock table:

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

Like in Section 4.4, we have shown the preservation Theorem 3 and we have shown that verified programs satisfy the following properties: null error freeness, partial correctness, and data race freeness.

### 5.6. Examples of Reasoning with Reentrant Locks

In this section, we show examples of reasoning with our verification system. We provide two examples: Section 5.6.1's class `Set` shows a typical use of reentrant locks as it often occurs in the Java library and Section 5.6.2 shows an advanced lock coupling example.

#### *5.6.1. A Typical Use of Reentrant Locks: class* `Set`

In the Java library, lock reentrancy is useful because container classes often feature client methods that are also helper methods. This happens if there is (1) a method which synchronizes on the receiver and is meant to be called by clients, but (2) this method can also be called by other methods of the same class. Because the other methods can also be synchronized on the receiver, lock reentrancy avoids to duplicate method implementation in two versions: a synchronized one (to be called by clients) and a lock free one (to be called by other methods of the class).

We present an example of the behavior described above for a class `Set` that represents mathematical sets. Internally, class `Set` is backed up by a list. Class `Set` contains a method `has` that should be used by clients to check if some element belongs to the receiver set. In addition, class `Set` contains method `add` which adds an element to the receiver set if this element is not already present. Both method `has` and method `add` lock the receiver set. Hence, as method `add` calls `has`, reentrant locks are crucial for class `Set`'s implementation.

First, we provide class `List` that backs up class `Set`. Class `List` is a *shallow* container: lists do not have permission to access their values. Values must be accessed by synchronizing on them. That is why lists ensure that they only contain initialized values (see predicate `state`'s implementation):

```
class List extends Object{
 Object element;
 List next;
 pred state = PointsTo(element,1,v) * v.initialized *
              PointsTo(next,1,n) * 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); }
   else{ result = false; }
  }
  result;
 }
 requires state * o.initialized; ensures state;
 void add(Object o){ List l = new List; l.init(o,this); }
}
```

We explain the meaning of class `List`'s predicate `state`. Predicate `state` gives access to field `next` of the list's first node (see `PointsTo(next,1,n)`) and to all `next` fields of subsequent nodes (because `state` is recursive, see `n.state`). In addition, predicate `state` (1) provides references to the values stored in the list (see `PointsTo(element,1,v)` and `n.state`) and (2) ensures that values are initialized (see `v.initialized` and `n.state`). It is crucial to ensure that values inside lists are initialized because predicate `state` does not give access to the values, it only provides references.

Second, we provide class `Set`. Class `Set` 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 `==`)[13]:

```
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();
   bool result = rep.has(o);
  unlock();
  result;
 }
```

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

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

*Remarks.* 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` or `add`, 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). As a result, lists do not need to be initialized (no `commit` statement in class `List`).

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

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` (see `init`'s proof outline below).

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)`). Method `add` in class `Set` could be specified similarly.

A simpler implementation of method `add` 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`), our implementation is still correct; while the simpler implementation could exhibit unexpected behaviors.

*Discussion.* Class `Set` exemplifies a typical use of lock reentrancy 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.

*5.6.2. Lock Coupling*

In this section, we illustrate how our verification system handles lock coupling. We use the following convenient abbreviations:

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

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 $n$.`count` == $n$.`next.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. [20] 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 $\pi$.`traversable`($\pi'$) (read as "if the current thread's lockset is $\pi'$, then the aggregate owned by object $\pi$ is traversable"). Concretely, this predicate says that no object owned by $\pi$ is contained in $\pi'$:

$\pi$.`traversable`($\pi'$) $\triangleq$
```
    (fa Object owner, Owned<owner> x)(!(π' contains x) | owner != π)
```

Note that in our definition of $\pi$.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 9 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 9'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<$\ell$>.

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

*Discussion.* 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*: with our specifications, nodes cannot be transferred from a list to another, because the nodes's owner would have to change. As a result, our solution works for a limited set of programs and further work is needed to handle all uses of fine-grained concurrency. In addition, because we use that our type system and our verification system are inter-dependent, automatically proving such a program would require a tool that combines reasoning about types and reasoning about separation logic formulas.

## 6. Related Work

The work that is closed related to our work is Parkinson's thesis [21]. 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 [28]), 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

```
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 9: A lock-coupling list

abstract predicates and permissions, he does not combine them as we do. Later, both Parkinson and Bierman [34] and Chin et al. [45] provided a flexible way to handle subclassing.

Separation-logic-based approaches for parallel programs [15, 46] focused on a theoretically elegant, but unrealistic, parallel operator. Notable exceptions are Hobor et al. [44] and Gotsman et al. [20] 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. [20]'s work, for instance the treatment of initialization of dynamically created locks. Our `initialized` predicate corresponds to what Gotsman calls lock handles (with his lock handle parameters corresponding to our class parameters). Since Gotsman's language supports deallocation of locks, he scales lock handles by fractional permissions in order to keep track of sharing. This is not necessary in a garbage-collected language. In addition to single-entrant locks, Gotsman also treats thread joining. We cover thread joining in a simpler and more powerful way, because we allow multiple readonly joining. The essential differences between Gotsman's and our paper are (1) that we treat reentrant locks, which are a different synchronization primitive than single-entrant locks, and (2) that we treat subclassing and extension of resource invariants in subclasses. Hobor et al.'s work [44] is very similar to [20].

A different approach is pursued by Vafeiadis, Parkinson et al. [47, 48, 49]. This work combines rely/guarantee 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; because they handle a larger class of problems. They do not treat reentrant locks.

Another related line of work is by Jacobs et al. [50] who extend the Boogie methodology for reasoning about object invariants [9] 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 pre-

vention, their programming discipline statically prevents reentrancy, although it may not be too hard to relax this at the cost of additional complexity. On the upside, Jacobs et al.'s verification system uses automatic standard SMT solvers, while – for automatic verification – we would require dedicated separation logic based provers.

Smans et al. [51, 52] 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 [53]). 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 [23].

Like Smans et al., Leino and Müller [54] presented a verification system for multithreaded programs that uses implicit dynamic frames and SMT solvers. Contrary to their previous work [50] 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.

In a more traditional approach, Ábráham, De Boer et al. [55, 56] apply assume-guarantee reasoning to a multithreaded Java-like language.

## 7. Conclusion

In this paper, we have presented a variant of permission-based separation logic that allows to reason 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 multiple simultaneous reads 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, and to apply it on a realistic programming language.

A first point for future work is to develop tool support for the existing logic. This involves several topics: (1) improving readability of the specification language, for example by extending an existing specification language such as JML [57]; (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 been investigated both by Tuerk [58] and Smans et al. [59], while the second topic has been investigated by Parkinson et al. [53]. However, in both cases the results have to be 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 5.6.2), uses class parameters to to model ownership. We will investigate how this can be done more systematically. We also plan to study whether permission annotations can be generated, instead of being written by the programmer.

In the longer term, we plan to study how the logic can be used in a more flexible way for concurrent data structures. In particular, specifications should be split into a functional and a concurrency part, in such a way that changing the locking policy or concurrency or synchronization primitives of an implementation would only affect validity of the concurrency specification, and not of the functional specification. Thus, if correctness of a program depends only on the functional specification of the data structure, then the change in the data structure's concurrency mechanism does not change correctness of the program. Eventually, this should also lead to a technique to reason about lock-free data structures, where some benign data races may be explicitly allowed by the logic.

## Acknowledgments

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

[1] W. Visser, K. Havelund, G. P. Brat, S. Park, F. Lerda, Model checking programs, Automated Software Engineering 10 (2) (2003) 203–232.

[2] M. Naik, A. Aiken, J. Whaley, Effective static race detection for Java, in: Programming Languages Design and Implementation, ACM Press, 2006, pp. 308–319.

[3] M. Naik, C. Park, K. Sen, D. Gay, Effective static deadlock detection, in: ICSE, 2009, pp. 386–396.

[4] C. Flanagan, S. Qadeer, A type and effect system for atomicity, in: Programming Languages Design and Implementation, Vol. 38 of ACM SIGPLAN Notices, ACM Press, 2003, pp. 338–349.

[5] M. Abadi, C. Flanagan, S. Freund, Types for safe locking: Static race detection for Java, ACM Transactions on Programming Languages and Systems 28 (2) (2006) 207–255.

[6] M. Huisman, Reasoning about Java programs in higher order logic using PVS and Isabelle, Ph.D. thesis, Computing Science Institute, University of Nijmegen (2001).

[7] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, R. Stata, Extended static checking for Java, in: Programming Languages Design and Implementation, Vol. 37, 2002, pp. 234–245.
URL `citeseer.ist.psu.edu/flanagan02extended.html`

[8] B. Beckert, R. Hähnle, P. H. Schmitt (Eds.), Verification of Object-Oriented Software: The KeY Approach, No. 4334 in Lecture Notes in Computer Science, Springer-Verlag, 2007.

[9] M. Barnett, R. DeLine, M. Fähndrich, K. R. M. Leino, W. Schulte, Verification of object-oriented programs with invariants, Journal of Object Technology 3 (6) (2004) 27–56.

[10] S. Owicki, D. Gries, An axiomatic proof technique for parallel programs, Acta Informatica Journal 6 (1975) 319–340.

[11] C. B. Jones, Tentative steps toward a development method for interfering programs, ACM Transactions on Programming Languages and Systems 5 (4) (1983) 596–619.

[12] C. A. R. Hoare, Towards a theory of parallel programming, in: Operating Systems Techniques, Academic Press, New York, NY, USA, 1972, pp. 61–71.

[13] C. A. R. Hoare, Monitors: an operating system structuring concept, Communications of the ACM 17 (10) (1974) 549–557.

[14] G. Andrews, Concurrent Programming: Principles and Practice, Benjamin/Cummings, 1991.

[15] P. W. O'Hearn, Resources, concurrency and local reasoning, Theoretical Computer Science 375 (1–3) (2007) 271–307.

[16] S. Ishtiaq, P. O'Hearn, BI as an assertion language for mutable data structures, in: Principles of Programming Languages, 2001, pp. 14–26.

[17] J. C. Reynolds, Separation logic: A logic for shared mutable data structures, in: Logic in Computer Science, IEEE Computer Society, 2002, pp. 55–74.

[18] S. Brookes, A semantics for concurrent separation logic., in: Conference on Concurrency Theory, Vol. 3170 of Lecture Notes in Computer Science, Springer-Verlag, 2004, pp. 16–34.

[19] R. Bornat, P. W. O'Hearn, C. Calcagno, M. Parkinson, Permission accounting in separation logic, in: J. Palsberg, M. Abadi (Eds.), Principles of Programming Languages, ACM Press, 2005, pp. 259–270.

[20] A. Gotsman, J. Berdine, B. Cook, N. Rinetzky, M. Sagiv, Local reasoning for storable locks and threads, in: Z. Shao (Ed.), Asian Programming Languages and Systems Symposium, Vol. 4807 of Lecture Notes in Computer Science, Springer-Verlag, 2007, pp. 19–37.

[21] M. Parkinson, Local reasoning for Java, Ph.D. thesis, University of Cambridge (2005).

[22] J. Boyland, Checking interference with fractional permissions, in: R. Cousot (Ed.), Static Analysis Symposium, Vol. 2694 of Lecture Notes in Computer Science, Springer-Verlag, 2003, pp. 55–72.

[23] C. Haack, C. Hurlin, Resource usage protocols for iterators, Journal of Object Technology 8 (4) (2009) 55–83.

[24] D. G. Clarke, J. M. Potter, J. Noble, Ownership types for flexible alias protection, in: ACM Conference on Object-Oriented Programming Systems, Languages, and Applications, Vol. 33:10 of ACM SIGPLAN Notices, ACM Press, New York, 1998, pp. 48–64.
URL `citeseer.ist.psu.edu/clarke98ownership.html`

[25] D. G. Clarke, S. Drossopoulou, Ownership, encapsulation and the disjointness of type and effect, in: ACM Conference on Object-Oriented Programming Systems, Languages, and Applications, ACM Press, 2002, pp. 292–310.

[26] P. Müller, Modular Specification and Verification of Object-Oriented Programs, Vol. 2262 of Lecture Notes in Computer Science, Springer-Verlag, 2002.

[27] R. DeLine, M. Fähndrich, Enforcing high-level protocols in low-level software, in: Programming Languages Design and Implementation, 2001, pp. 59–69.

[28] C. Haack, C. Hurlin, Separation logic contracts for a Java-like language with fork/join, in: J. Meseguer, G. Rosu (Eds.), Algebraic Methodology and Software Technology, Vol. 5140 of Lecture Notes in Computer Science, Springer-Verlag, 2008, pp. 199–215.

[29] C. Haack, M. Huisman, C. Hurlin, Reasoning about Java's reentrant locks, in: G. Ramalingam (Ed.), Asian Programming Languages and Systems Symposium, Vol. 5356 of Lecture Notes in Computer Science, Springer-Verlag, 2008, pp. 171–187.

[30] C. Hurlin, Specification and verification of multithreaded object-oriented programs with separation logic, Ph.D. thesis, Université Nice Sophia Antipolis (2009).

[31] K. Crary, D. Walker, G. Morrisett, Typed memory management in a calculus of capabilities, in: Principles of Programming Languages, 1999, pp. 262–275.

[32] F. Smith, D. Walker, G. Morrisett, Alias types, in: G. Smolka (Ed.), European Symposium on Programming, Vol. 1782 of Lecture Notes in Computer Science, Springer-Verlag, 2000, pp. 366–381.

[33] L. Jia, D. Walker, ILC: A foundation for automated reasoning about pointer programs, in: European Symposium on Programming, 2006, pp. 131–145.

[34] M. Parkinson, G. Bierman, Separation logic, abstraction and inheritance, in: Principles of Programming Languages, ACM Press, 2008, pp. 75–86.

[35] M. Parkinson, G. Bierman, Separation logic and abstraction, in: J. Palsberg, M. Abadi (Eds.), Principles of Programming Languages, ACM Press, 2005, pp. 247–258.

[36] A. Igarashi, B. Pierce, P. Wadler, Featherweight Java: a minimal core calculus for Java and GJ, ACM Trans. Program. Lang. Syst. 23 (3) (2001) 396–450.

[37] C. Haack, C. Hurlin, Separation logic contracts for a Java-like language with fork/join, Tech. Rep. 6430, INRIA (Jan. 2008).

[38] P. Wadler, A taste of linear logic., in: Mathematical Foundations of Computer Science, 1993, pp. 185–210.

[39] P. W. O'Hearn, D. J. Pym, The logic of bunched implications, Bulletin of Symbolic Logic 5 (2) (1999) 215–244.

[40] P. W. O'Hearn, J. Reynolds, H. Yang, Local reasoning about programs that alter data structures, in: L. Fribourg (Ed.), Computer Science Logic, Vol. 2142 of Lecture Notes in Computer Science, Springer-Verlag, 2001, pp. 1–19, invited paper.

[41] R. Bornat, C. Calcagno, H. Yang, Variables as resource in separation logic, in: Mathematical Foundations of Programming Semantics, Vol. 155 of Electronic Notes in Theoretical Computer Science, Elsevier, 2005, pp. 247–276.

[42] S. Microsystems, Java's documentation: `http://java.sun.com/`.

[43] J. Boyland, Semantics of fractional permissions with nesting, Tech. rep., University of Wisconsin at Milwaukee (Dec. 2007).

[44] A. Hobor, A. Appel, F. Nardelli, Oracle semantics for concurrent separation logic, in: S. Drossopoulou (Ed.), Programming Languages and Systems: Proceedings of the 17th European Symposium on Programming, ESOP 2008, Vol. 4960 of Lecture Notes in Computer Science, Springer-Verlag, 2008, pp. 353–367.

[45] W. Chin, C. David, H. Nguyen, S. Qin, Enhancing modular OO verification with separation logic, in: G. C. Necula, P. Wadler (Eds.), Principles of Programming Languages, ACM Press, 2008, pp. 87–99.

[46] J. Berdine, C. Calcagno, P. W. O'Hearn, Smallfoot: Modular automatic assertion checking with separation logic, in: F. S. de Boer, M. M. Bonsangue, S. Graf, W.-P. de Roever (Eds.), Formal Methods for Components and Objects, Vol. 4111 of Lecture Notes in Computer Science, Springer-Verlag, 2005, pp. 115–137.

[47] V. Vafeiadis, M. J. Parkinson, A marriage of rely/guarantee and separation logic, in: L. Caires, V. T. Vasconcelos (Eds.), Conference on Concurrency Theory, Vol. 4703 of Lecture Notes in Computer Science, Springer-Verlag, 2007, pp. 256–271.

[48] M. Dodds, X. Feng, M. Parkinson, V. Vafeiadis, Deny-guarantee reasoning, in: European Symposium on Programming, Lecture Notes in Computer Science, Springer-Verlag, 2009, pp. 363–377.

[49] J. Wickerson, M. Dodds, M. J. Parkinson, Explicit stabilisation for modular rely-guarantee reasoning, in: A. D. Gordon (Ed.), European Symposium on Programming, Vol. 6012 of Lecture Notes in Computer Science, Springer-Verlag, 2010, pp. 610–629.

[50] B. Jacobs, J. Smans, F. Piessens, W. Schulte, A statically verifiable programming model for concurrent object-oriented programs., in: International Conference on Formal Engineering Methods, 2006, pp. 420–439.

[51] J. Smans, B. Jacobs, F. Piessens, W. Schulte, An automatic verifier for Java-like programs based on dynamic frames, in: J. L. Fiadeiro, P. Inverardi (Eds.), Fundamental Approaches to Software Engineering, Vol. 4961 of Lecture Notes in Computer Science, Springer-Verlag, 2008, pp. 261–275.

[52] J. Smans, B. Jacobs, F. Piessens, Implicit dynamic frames: Combining dynamic frames and separation logic, in: S. Drossopoulou (Ed.), European Conference on Object-Oriented Programming, Vol. 5653 of Lecture Notes in Computer Science, Springer-Verlag, 2009, pp. 148–172.

[53] D. DiStefano, M. Parkinson, jStar: Towards practical verification for Java, in: ACM Conference on Object-Oriented Programming Systems, Languages, and Applications, ACM Press, 2008, pp. 213–226.

[54] K. R. M. Leino, P. Müller, A basis for verifying multi-threaded programs, in: G. Castagna (Ed.), European Symposium on Programming, Vol. 5502 of Lecture Notes in Computer Science, Springer-Verlag, 2009, pp. 378–393.

[55] E. Ábrahám, F. S. de Boer, W.-P. de Roever, M. Steffen, Tool-supported proof system for multithreaded Java, in: F. S. de Boer, M. M. Bonsangue, S. Graf, W.-P. de Roever (Eds.), Formal Methods for Components and

Objects, No. 2852 in Lecture Notes in Computer Science, Springer-Verlag, 2003, pp. 1–32.

[56] F. S. de Boer, A sound and complete shared-variable concurrency model for multi-threaded Java programs, in: International Conference on Formal Methods for Open Object-based Distributed Systems, 2007, pp. 252–268.

[57] G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. R. Cok, P. Müller, J. Kiniry, P. Chalin, JML Reference Manual, department of Computer Science, Iowa State University. Available from `http://www.jmlspecs.org` (Feb. 2007).

[58] T. Tuerk, A formalisation of smallfoot in HOL, in: S. Berghofer, T. Nipkow, C. Urban, M. Wenzel (Eds.), Theorem Proving in Higher-Order Logics, Vol. 5674 of Lecture Notes in Computer Science, Springer-Verlag, 2009, pp. 469–484.

[59] J. Smans, B. Jacobs, F. Piessens, Heap-dependent expressions in separation logic, submitted.

# Appendix A. Auxiliary Definitions

## Appendix A.1. Definitions of lookup functions

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

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

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

**Axiom Lookup,** $\mathsf{axiom}(t\texttt{<}\bar{\pi}\texttt{>}) = F$**:**

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

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

(Ax Class)
$$\frac{\texttt{class}\ C\texttt{<}\bar{T}\ \bar{\alpha}\texttt{>}\ \texttt{ext}\ U\ \texttt{impl}\ \bar{V}\ \texttt{\{}fd\texttt{*}\ pd\texttt{*}\ ax\texttt{*}\ md\texttt{*\}}}{\mathsf{axiom}(C\texttt{<}\bar{\pi}\texttt{>}) = \mathsf{axiom}(ax\texttt{*}[\bar{\pi}/\bar{\alpha}]) * \mathsf{axiom}((U, \bar{V})[\bar{\pi}/\bar{\alpha}])}$$

(Ax Interface)
$$\frac{\texttt{interface}\ I\texttt{<}\bar{T}\ \bar{\alpha}\texttt{>}\ \texttt{ext}\ \bar{U}\ \texttt{\{}pt\texttt{*}\ ax\texttt{*}\ mt\texttt{*\}}}{\mathsf{axiom}(I\texttt{<}\bar{\pi}\texttt{>}) = \mathsf{axiom}(ax\texttt{*}[\bar{\pi}/\bar{\alpha}]) * \mathsf{axiom}(\bar{U}[\bar{\pi}/\bar{\alpha}])}$$

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.

- In mtype, we replace the implicit result-parameter `result` by an explicit existential quantifier over the postcondition. This is technically convenient for the theory.

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

(Mlkup `Object`)

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

(Mlkup Defn)

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

(Mlkup Inherit) $\quad m \notin \mathsf{dom}(md\texttt{*})$

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

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

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

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

(Mtype Interface)

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

(Mtype Interface Inherit) $\quad \texttt{interface}\ I\texttt{<}\bar{T}\ \bar{\alpha}\texttt{>}\ \texttt{ext}\ \bar{U}, V, \bar{U}'\ \texttt{\{}pt\texttt{*}\ ax\texttt{*}\ mt\texttt{*\}}$

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

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

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

Remarks on predicate lookup:

- The "ext `Object`" in $\mathsf{plkup}(\texttt{init}, \texttt{Object})$ 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`), $\mathsf{df}(T)$ is the default value of type $T$ ($\mathsf{df}$ is formally defined in Section 2.2).

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

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

(Plkup `Object`)

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

(Plkup Defn)

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

(Plkup `init`)

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

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

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

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

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

(Ptype Interface)

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

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

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

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

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

The partial function $\mathsf{ptype}(P, t\texttt{<}\bar{\pi}\texttt{>})$ is extended to predicate selectors $P\texttt{@}C$ as follows:

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

### Appendix 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:

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

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

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

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

$$[\![op]\!] \quad \in \quad \mathsf{Heap} \rightarrow \bigcup_{(\bar{T}, U)\ \in\ \mathsf{type}(op)} [\![\bar{T}]\!] \rightarrow [\![U]\!]$$

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:

$$[\![C\ \texttt{classof}]\!]^h(o) \triangleq \begin{cases} \texttt{true} & \text{if } o \neq \texttt{null} \text{ and } h(o)_1 = C\texttt{<}\bar{\pi}\texttt{>} \text{ for some } \bar{\pi} \\ \texttt{false} & \text{if } o \neq \texttt{null},\ h(o)_1 = D\texttt{<}\bar{\pi}\texttt{>}, \text{ and } D \neq C \\ \texttt{false} & \text{if } o = \texttt{null} \\ \mathsf{undef} & o \notin \mathsf{dom}(h) \end{cases}$$

$$[\![o\ \texttt{instanceof}\ T]\!]^h \triangleq \begin{cases} \texttt{true} & \text{if } o \neq \texttt{null} \text{ and } h(o)_1 <: T \\ \texttt{false} & \text{if } o \neq \texttt{null} \text{ and } h(o)_1 \not<: T \\ \texttt{false} & \text{if } o = \texttt{null} \\ \mathsf{undef} & \text{if } o \notin \mathsf{dom}(h) \end{cases}$$

Formally, the semantics of operators is expressed as follows:

**Semantics of Operators:** $[\![op(\bar{v})]\!] : \mathsf{Heap} \to \mathsf{Stack} \rightharpoonup \mathsf{ClVal}$**:**

(Sem Op)
$$\frac{[\![w_1]\!]^h_s = v_1 \quad \cdots \quad [\![w_n]\!]^h_s = v_n \quad [\![op]\!]^h(v_1, \ldots, v_n) = v}{[\![op(w_1, \ldots, w_n)]\!]^h_s = v}$$

### Appendix A.3. Typing rules

*Rules for Section 3* 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 $\mathsf{ObjId} \cup \mathsf{Var} \rightharpoonup \mathsf{Type}$. We use the meta-variable $\Gamma$ to range over type environments. $\Gamma_{\mathsf{hp}}$ denotes the *restriction of* $\Gamma$ *to* ObjId:

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

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

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

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

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)
$$\frac{T \in \{\texttt{void}, \texttt{int}, \texttt{bool}, \texttt{perm}\}}{\Gamma \vdash T : \diamond}$$

(Ty Ref)
$$\frac{t\texttt{<}\bar{T}\ \bar{\alpha}\texttt{>} \in ct \quad (\exists \Gamma' \subset \Gamma)(\Gamma' \vdash \diamond \quad \Gamma' \vdash \bar{\pi} : \bar{T}[\bar{\pi}/\bar{\alpha}])}{\Gamma \vdash t\texttt{<}\bar{\pi}\texttt{>} : \diamond}$$

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

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

As models of formulas are tuples that contain a heap and a stack (see Section 3.1.2), we define a well-typedness judgment for objects, heaps, and stacks:

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

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

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

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

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

We now define well-typed heaps and stacks:

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

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

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

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 $\Gamma \vdash \pi : T$:**

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

$$\text{(Val Sub)}\quad \frac{\Gamma \vdash \pi : T \quad T <: U}{\Gamma \vdash \pi : U} \qquad \text{(Val Null)}\quad \frac{\Gamma \vdash t\texttt{<}\bar{\pi}\texttt{>} : \diamond}{\Gamma \vdash \texttt{null} : t\texttt{<}\bar{\pi}\texttt{>}}$$

$$\text{(Val Int)}\quad \frac{\Gamma \vdash \diamond}{\Gamma \vdash n : \texttt{int}} \qquad \text{(Val Bool)}\quad \frac{\Gamma \vdash \diamond}{\Gamma \vdash b : \texttt{bool}} \qquad \text{(Val Full)}\quad \frac{\Gamma \vdash \diamond}{\Gamma \vdash 1 : \texttt{perm}} \qquad \text{(Val Split)}\quad \frac{\Gamma \vdash \pi : \texttt{perm}}{\Gamma \vdash \texttt{split}(\pi) : \texttt{perm}}$$

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

$$\text{(Exp Sub)}\quad \frac{\Gamma \vdash e : T \quad T <: U}{\Gamma \vdash e : U} \qquad \text{(Exp Var)}\quad \frac{\Gamma \vdash \diamond \quad \Gamma(\ell) = T}{\Gamma \vdash \ell : T}$$

$$\text{(Exp Op)}\quad \frac{\Gamma \vdash \bar{e} : \bar{U} \quad \mathsf{type}(op)(\bar{U}) = T}{\Gamma \vdash op(\bar{e}) : T}$$

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

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

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

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

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

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

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

*Rules for Section 4* To cover Section 4's `Join` formula, we extend the judgment for well-typed formulas as follows:

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

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

*Rules for Section 5* To accommodate Section 5.3's `lockset`'s type, we update the previous typing rule for good types:

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

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

The following typing rule extends typing to values representing locksets:

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

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

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

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

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

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

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