Separation Logic Contracts for a Java-like Language with Fork/Join

Christian Haack[1]* and Clément Hurlin[2]**

[1] Radboud Universiteit Nijmegen, The Netherlands
[2] INRIA Sophia Antipolis - Méditerranée, France

Abstract. We adapt a variant of permission-accounting separation logic to a concurrent Java-like language with fork/join. To support both concurrent reads and information hiding, we combine fractional permissions with abstract predicates. As an example, we present a separation logic contract for iterators that prevents data races and concurrent modifications. Our program logic is presented in an algorithmic style: we avoid structural rules for Hoare triples and formalize logical reasoning about typed heaps by natural deduction rules and a set of sound axioms. We show that verified programs satisfy the following properties: data race freedom, absence of null-dereferences and partial correctness.

1 Introduction

1.1 Context

Over the past ten years or so, substructural logics and type systems have proven to be very valuable formalisms for reasoning about pointer-manipulating programs. Examples include static capabilities [10,11], alias types [29] and separation logic [18,28]. In these systems, the underlying specification language contains linear formulas for specifying memory access policies. Whereas traditional program logics control memory access via frame conditions, separation logic tightly integrates access policy specifications into the formula language itself. Formulas represent access tickets to heap space, and possession of access tickets gets verified statically. Access policies are tightly coupled with assertions about memory content, so that separation logic's Hoare rules make it impossible to maintain assertions that can be invalidated by thread interference or memory updates through unknown aliases. This is achieved without annoying side conditions like non-interference tests or frame conditions.

While initially separation logic mostly focused on low level programs, researchers have more recently started to adapt it to object-oriented features for use in contract languages for OO [25,26], and very recently [9,27].

1.2 Contributions

We present the careful design of a small Java-like model language with separation logic contracts, including the definition of a program logic and its soundness proof. Our language has simple threads, with fork/join as concurrency primitives. In order to facilitate concurrent reads we employ fractional permissions [5]. Our rules allow multiple

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

threads to join on the same thread, in order to read-share the dead thread's resources. This is not possible with a lexically scoped parallel composition operator or with Posix threads, and is thus not supported by recent work that adapts separation logic to Posix threads [14]. To support data abstraction and recursive data types, we use abstract predicates [26]. *Class axioms* complement abstract predicates to export relations between predicates without revealing their full definitions. Abstract predicates satisfying a split/merge axiom generalize datagroups [21], which are common in specification languages for OO. In order to support concurrent read access to whole datagroups (rather than single fields), access permission to datagroups can be split by splitting their permission parameters. In order to allow fine-grained permission-splitting for overlapping datagroups, we support datagroups with multiple permission parameters. To achieve modular soundness in the presence of subclassing, we axiomatize the "stack of class frames" [12,1] in separation logic. We support *value-parametrized classes*, where class parameters have the same purpose as `final` ghost fields in specification languages like JML [20]. In particular, class parameters can represent static ownership relations.

1.3 Background on Separation Logic and Fractional Permissions

Separation logic combines the usual logical operators with the points-to predicate $x.f \mapsto v$, the resource conjunction $F * G$, and the resource implication $F \mathrel{-\!\!*} G$.

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

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

The crucial difference to standard Hoare logic is that both these rules have a precondition of the form $x.f \mapsto _$.[3] 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 $F * F$. The resource implication $\mathrel{-\!\!*}$ matches the resource conjunction $*$, in the sense that the modus ponens law is satisfied: $F * (F \mathrel{-\!\!*} G)$ implies G. However, $F * (F \mathrel{-\!\!*} G)$ *does not* imply $F * G$. In English, $F \mathrel{-\!\!*} G$ is pronounced as "consume F yielding G". In terms of tickets, $F \mathrel{-\!\!*} G$ permits to trade ticket F and receive ticket G in return.

Separation logic is particularly useful for concurrent programs: two concurrent threads simply split the resources that they may access, as formalized by the rule for the parallel composition $t \mid t'$ of threads t and t' [22].

$$\frac{\{F\}t\{G\} \quad \{F'\}t'\{G'\}}{\{F * F'\}t \mid t'\{G * G'\}}$$

With this concurrency rule, separation logic prevents data races. There is a caveat, though. The rule does not allow concurrent reads. Boyland [5] solved this problem with a very intuitive idea, which was later adapted to separation logic [4]. The idea is that (1) *access tickets are splittable*, (2) *a split of an access ticket still grants read access* and (3) *only a whole access ticket grants write access*. To account for multiple

[3] $x.f \mapsto _$ is short for $(\exists v)(x.f \mapsto v)$.

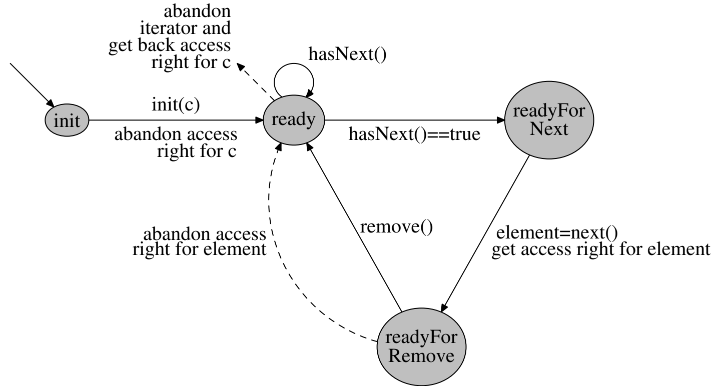

Fig. 1. Usage Protocol for Iterators

splits, Boyland uses fractions, hence the name *fractional permissions*. In permission-accounting separation logic [4], access tickets $x.f \mapsto v$ are superscripted by fractions π. $x.f \overset{\pi}{\longmapsto} v$ is equivalent to $x.f \overset{\pi/2}{\longmapsto} v * x.f \overset{\pi/2}{\longmapsto} v$. In the Hoare rules, writing requires the full fraction 1, whereas reading just requires *some* fraction π:

$$\{x.f \overset{1}{\longmapsto} _ * F\}x.f\texttt{=}v\{x.f \overset{1}{\longmapsto} v * F\} \qquad \{x.f \overset{\pi}{\longmapsto} v * F\}y\texttt{=}x.f\{x.f \overset{\pi}{\longmapsto} v * v \texttt{==} y * F\}$$

Permission-accounting separation logic maintains the global invariant that the sum of all fractional permissions to the same cell is always at most 1. This prevents read-write and write-write conflicts, but permits concurrent reads.

In our Java-like language, we use ASCII and write `Perm`$(x.f,\pi)$ for $x.f \overset{\pi}{\longmapsto} _$, and `PointsTo`$(x.f,\pi,v)$ for $x.f \overset{\pi}{\longmapsto} v$.

1.4 Example: A Usage Protocol for Iterators

Often one wants to constrain object clients to adhere to certain usage protocols. Object usage protocols can, for instance, be specified in typestate systems [12] or, using ghost fields, in general purpose specification languages. A limitation of these techniques is that state transitions must always be associated with method calls. This is sometimes not sufficient. Consider for instance a variant of Java's `Iterator` interface (enriched with an `init` method to avoid constructor contracts):

```
interface Iterator {
  void init(Collection c);
  boolean hasNext();
  Object next();
  void remove();
}
```

If iterators are used in an undisciplined way, there is the danger of unwanted concurrent modification of the underlying collection (both of the collection elements and the collection itself). Moreover, in concurrent programs bad iterator usage can result in data races. It is therefore important that `Iterator` clients adhere to a usage discipline. Figure 1 shows a state machine that defines a safe iterator usage discipline. Unfortunately, the dashed transitions are not supported by existing typestate systems, because they are not associated with method calls. Specifying

this protocol with classical program logics would be clumsy. In [13] (Section 1.1.4), Girard explains how *linear implications can be used to logically represent state transitions*. Applying this idea to the iterator protocol, we obtain the following formalization where the dashed transitions are represented by resource implications:

```
interface Iterator<perm p, Collection iteratee> {
   pred ready;                          // prestate for iteration cycle
   pred readyForNext;                   // prestate for next()
   pred readyForRemove<Object element>; // prestate for remove()
   axiom ready -* iteratee.state<p>;               // stop iterating
   axiom readyForRemove<e> * e.state<p> -* ready;  // back to ready
   req init * c.state<p> * c==iteratee; ens ready;
   void init(Collection c);
   req ready; ens ready & (result -* readyForNext);
   boolean hasNext();
   req readyForNext; ens result.state<p> * readyForRemove<result>;
   Object next();
   req readyForRemove<_> * p==1; ens ready;
   void remove();
}
```

The interface has two parameters: firstly, a permission p and, secondly, the `iteratee`. If the permission parameter is instantiated by a fraction $p < 1$, one obtains a read-only iterator, otherwise a read-write iterator. The states of our our state diagram are represented by three abstract predicates: `ready`, `readyForNext` and `readyForRemove`.

Class axioms express relations between abstract predicates, without revealing the complete predicate definitions. Implementations must define the abstract predicates by separation logic formulas such that the class axioms are tautologically true. In the example, the two class axioms represent the dashed transitions of the state machine. We represent the heap space associated with an object by a generic datagroup `state`, which has a default definition in the `Object` class and needs to be overridden by each class. The definition of this `state` predicate should describe the heap space associated with the object. Often this heap space will consist of the object's fields only, but sometimes it will also include other objects and change dynamically, as in the case of `Collection` objects. The `state` predicate is parametrized by a fraction so that it can be read-shared.

The precondition of `init()` consumes a fraction p of the access right for the iteratee and puts the iterator in the `ready` state. The crux is that, by linearity, the iterator client temporarily looses a p-fraction of the access right on the collection, which he can only gain back by "invoking" the first class axiom. The `init` predicate in `init()`'s precondition is a special abstract predicate that every object enters right after object creation and that grants access to all of the object's fields.

The postcondition of `hasNext()` uses a resource implication whose antecedent is a boolean expression. We treat boolean expressions as *copyable resources* that satisfy $e \mathrel{-\!\!*} (e * e)$.[4] Furthermore, `hasNext()`'s postcondition uses *additive conjunction* `&`. A

[4] We could equivalently use classical implication: `result` ⇒ `readyForNext`. Even the two class axioms could equivalently use classical implication, because they are tautologies, and in

resource satisfies F & G, if it satisfies both F and G.[5] Operationally, & represents *choice*. If F & G holds, then F and G are available, but are interdependent: using either one of them destroys the other one, too. Additive conjunction can conveniently represent non-deterministic state transitions, as exhibited in `hasNext()`'s postcondition. Note that this postcondition allows clients to stay in the `ready`-state, even if `hasNext()==true`. This can, for instance, be useful for removing the 10th element of an ordered collection.

In our companion report [16], we have implemented the `Iterator` interface for a doubly linked list implementation of the `Collection` interface. In [15], we refine the protocol to support unrestricted access to immutable collection elements, and to support shallow collections that do not govern access to their elements.

1.5 Example: Representing Datagroups

We represent *datagroups* [21] as abstract predicates satisfying a datagroup axiom that says *split/merging datagroup parameters split/merges datagroups*:

$$\texttt{group}\ P\texttt{<}\bar{T}\ \bar{x}\texttt{>;}\ \stackrel{\Delta}{=}\ \begin{array}{l}\texttt{pred}\ P\texttt{<}\bar{T}\ \bar{x}\texttt{>; axiom}\ P\texttt{<}\bar{x}\texttt{>}\ \texttt{*-*}\ \texttt{(}P\texttt{<}\bar{e}\texttt{>} * P\texttt{<}\bar{e}\texttt{>)};\\ \text{where } e_i \stackrel{\Delta}{=} x_i/2,\ \text{if } T_i = \texttt{perm},\ \text{and } e_i \stackrel{\Delta}{=} x_i,\ \text{otherwise}\end{array}$$

The formula F `*-*` G is short for `(`F `-*` G`)` & `(`G `-*` F`)`. Here are simple examples of a legal and an illegal datagroup definition (where `|` is disjunction):

```
group P<perm p> = Perm(this.f,p) * Perm(this.g,p);
```
legal because the datagroup axiom holds
```
group P<perm p> = Perm(this.f,p) | Perm(this.g,p);
```
illegal because the datagroup axiom's right-to-left direction does not hold

On the right, you see a fractional permission version of Leino's running example [21]. The datagroups `position` and `color` are nested in `state`, as expressed by the two class axioms. The formula "F `ispartof` G" is a derived form for G `-*` `(`F `*` `(`F `-*` G`))`. Intuitively, this formula says that F is a physical part of G: one can take G apart into F and its complement F `-*` G, and one can put the two parts back together to obtain G back.

```
interface Sprite {
   group position<perm p>;
   group color<perm p>;
   axiom position<p> ispartof state<p>;
   axiom color<p> ispartof state<p>;
   req position<1>; ens position<1>;
   void updatePosition();
   req color<1>; ens color<1>;
   void updateColor();
   req state<1>; ens state<1>;
   void update();
   req state<p>; ens state<p>;
   void display();
}
```

intuitionistic separation logic F `-*` G is a tautology if and only if $F \Rightarrow G$ is. However, in our implementations of the `Iterator` interface, we use true resource implications that cannot be replaced by classical implications. In this paper, we avoid classical implication because having just one implication simplifies the natural deduction rules for reasoning about resources.

[5] In contrast, a resource satisfies $F * G$ if it can be split into separate resources, one of which satisfies F and the other satisfies G.

1.6 Example: Recursive and Overlapping Datagroups

Our next example illustrates that multiple threads can concurrently access overlapping datagroups, as long as they only read-access their intersection. Consider a linked list that implements a simple class roster. Each node stores a student identifier and a grade. We design the roster interface so that multiple threads can concurrently read the roster. Moreover, when a thread updates the grades we allow other threads to concurrently read the student identifiers. To this end, the interface defines two datagroups `ids_and_links<p,q>` and `grades_and_links<p,q>` that overlap in the links of the list. The permission parameter `p` is associated with the student id fields and grade fields, respectively. The permission parameter `q` is associated with the links.

```
interface Roster {
   group ids_and_links<perm p, perm q>;
   group grades_and_links<perm p, perm q>;
   axiom state<p> *-* (ids_and_links<p,p/2> * grades_and_links<p,p/2>);
   req grades_and_links<1,p> * ids_and_links<q,r>;
   ens grades_and_links<1,p> * ids_and_links<q,r>;
   void updateGrade(int id, int grade);
   req ids_and_links<p,q>; ens ids_and_links<p,q>;
   bool contains(int id);
}
```

The `updateGrade()` method requires write access (permission `1`) for the grades and read access for the links and ids. The `contains()` method requires read permission for the ids and the links. The axiom exposes that the `state` datagroup is the union of the datagroups `ids_and_links` and `grades_and_links` and that these datagroups overlap on the links. In our companion report [16], we have implemented this interface.

2 A Model Language with Separation Logic Contacts

2.1 Syntax

We distinguish between read-only variables ι, read-write variables ℓ, and logic variables α. Method parameters (including `this`) are read-only. Logic variables can only occur in specifications and types. They range over both fractional permissions and values (like integers, object identifiers and `null`).

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

We include read-only variables (but not read-write variables) in the syntax domain of *values*. This is convenient for our substitution-based operational semantics. *Fractional permissions* are represented symbolically: $\mathtt{split}^n(1)$ represents the concrete fraction $\frac{1}{2^n}$. In examples, we sometimes write $\frac{1}{2^n}$ as syntax sugar for $\mathtt{split}^n(1)$. *Specification values* union values and fractional permissions. Interfaces and classes are parametrized by specification values. Correspondingly, object types $t\texttt{<}\bar{\pi}\texttt{>}$ instantiate the parameters.

$n \in \mathsf{Int}$	integers	$b \in \mathsf{Bool} = \{\mathtt{true}, \mathtt{false}\}$	booleans
$u,v,w \in \mathsf{Val}$	$::= \mathtt{null} \mid n \mid b \mid o \mid \iota$		values
$\pi \in \mathsf{SpecVal}$	$::= v \mid 1 \mid \mathtt{split}(\pi) \mid \alpha$		specification values
$T,U,V,W \in \mathsf{Ty}$	$::= \mathtt{void} \mid \mathtt{int} \mid \mathtt{bool} \mid t\mathtt{<}\bar{\pi}\mathtt{>} \mid \mathtt{perm}$		types

Interface Declarations:

$F \in \mathsf{Formula} ::= \ldots$	specification formulas (defined in Section 2.3)
$spec ::= \mathtt{req}\ F; \mathtt{ens}\ F;$	pre- and postconditions
$mt ::= \mathtt{<}\bar{T}\ \bar{\alpha}\mathtt{>}\ spec\ U\ m(\bar{V}\ \bar{\iota})$	method types (scope of $\bar{\alpha}, \bar{\iota}$ is $\bar{T}, spec, U, \bar{V}$)
$pt ::= \mathtt{pred}\ P\mathtt{<}\bar{T}\ \bar{\alpha}\mathtt{>}$	predicate types
$ax ::= \mathtt{axiom}\ F$	class axioms
$int \in \mathsf{Interface} ::= \mathtt{interface}\ I\mathtt{<}\bar{T}\ \bar{\alpha}\mathtt{>}\ \mathtt{ext}\ \bar{U}\ \{pt{*}\ ax{*}\ mt{*}\}$	
	interfaces (scope of $\bar{\alpha}$ is $\bar{T}, \bar{U}, pt{*}, ax{*}, mt{*}$)

Syntactic restriction: The type "`perm`" may only occur inside angle brackets or formulas.

Method types include pre- and postconditions and are parametrized by logic variables. In examples, we often leave these quantifiers over logic variables implicit. Interfaces may declare *abstract predicates* and classes must implement them by providing concrete definitions as separation logic formulas. Like [26], we allow abstract predicates to have parameters in addition to the implicit self-parameter (as listed in the typed formal parameter lists $\bar{T}\ \bar{\alpha}$). The types $\bar{T}$ for predicate parameters range over all Java types and the distinguished type `perm` for fractional permissions.

We assume that the `Object` class declares a distinguished datagroup called `state`:

```
class Object { group state<perm p> = true; }
```

This datagroup represents the access permissions for the *object state*. Every class must extend it and thereby define what the object states of its instances are. Our syntax for predicate extensions is as follows:

```
class C ext D { ... ext pred P<T̄ x̄> by F;  ... }
```

Semantically, the extension F of abstract predicate P gets $*$-*conjoined* with P's definition in C's superclass D. We do not allow arbitrary predicate redefinitions in subclasses in order to facilitate modular verification, avoiding re-verification of inherited methods.

Class Declarations:

$fd ::= T\ f$	field declarations
$pd ::=$	predicate definitions
$\quad \mathtt{final?}\ \mathtt{pred}\ P\mathtt{<}\bar{T}\ \bar{\alpha}\mathtt{>}\mathtt{=}F$	$\quad$ root definition (scope of $\bar{\alpha}$ is F)
$\quad \mathtt{final?}\ \mathtt{ext}\ \mathtt{pred}\ P\mathtt{<}\bar{T}\ \bar{\alpha}\mathtt{>}\ \mathtt{by}\ F$	$\quad$ extension (scope of $\bar{\alpha}$ is F)
$md ::= \mathtt{final?}\ \mathtt{<}\bar{T}\ \bar{\alpha}\mathtt{>}\ spec\ U\ m(\bar{V}\ \bar{\iota})\{c\}$	method (scope of $\bar{\alpha}, \bar{\iota}$ is $\bar{T}, spec, U, \bar{V}, c$)
$cl \in \mathsf{Class} ::= \mathtt{final?}\ \mathtt{class}\ C\mathtt{<}\bar{T}\ \bar{\alpha}\mathtt{>}\ \mathtt{ext}\ U\ \mathtt{impl}\ \bar{V}\ \{fd{*}\ pd{*}\ ax{*}\ md{*}\}$	
	class (scope of $\bar{\alpha}$ is $\bar{T}, U, \bar{V}, fd{*}, pd{*}, ax{*}, md{*}$)
$ct \subseteq \mathsf{Interface} \cup \mathsf{Class}$	class tables

Syntactic restrictions:
- The type "`perm`" may only occur inside angle brackets or specification formulas.
- Cyclic predicate definitions in *ct* must be positive.

The *first syntactic restriction* ensures that fractional permissions do not spill into the executable part of the language. The *second syntactic restriction* ensures that predicate implementations (which can be recursive) are well-founded. We allow negative dependencies of predicate P on predicate Q as long as Q does not also depend on P.

We use the symbol $\preceq_{ct}$ for the partial order on type identifiers induced by class table ct, usually leaving the subscript ct implicit. We write $s \prec_1 t$ when s and t are neighbours with respect to $\preceq$. *Subtyping* is inductively defined by the following rules:

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

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

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

```
class Thread ext Object {
  final void fork(); final void join();
  req false; ens true; void run() { null }
}
```

The `run`-method is meant to be overridden. The contracts for `fork` and `join` are omitted, because our verification system ignores them anyway. Instead, it uses the precondition for `run` as the precondition for `fork` and the postcondition for `run` as the postcondition for `join`. The methods `fork` and `join` do not have implementations, but the operational semantics treats them in a special way[6]: o.`fork()` creates a new thread, whose thread identifier is o, and executes o.`run()` in this thread. The o.`fork`-method should not be called more than once (on the same receiver o). A second call results in blocking. o.`join()` blocks until thread o has terminated.

Commands:

$op \in \mathsf{Op} \supseteq \{\texttt{==}, \texttt{!}, \texttt{\&}, \texttt{|}\} \cup \{C\ \texttt{isclassof} \mid C \in \mathsf{ClassId}\}$

$c \in \mathsf{Cmd} ::=$ commands

v	return value (or `null` in case of type `void`)
$T\ \ell; c$	local variable declaration (scope of ℓ is c)
$\texttt{final}\ T\ \iota\texttt{=}\ell; c$	local read-only variable declaration (scope of ι is c)
$\texttt{unpack}\ \texttt{(ex}\ T\ \alpha\texttt{)}\ \texttt{(}F\texttt{)}; c$	unpacking an existential (scope of α is F, c)
$hc; c$	first do hc, then do c

$hc \in \mathsf{HeadCmd} ::= \ell\texttt{=}v \mid \ell\texttt{=}op(\bar{v}) \mid \ell\texttt{=}v.f \mid v.f\texttt{=}v \mid \ell\texttt{=(}T\texttt{)}v \mid \ell\texttt{=new}\ C\texttt{<}\bar{\pi}\texttt{>} \mid$
$\qquad \texttt{if (}v\texttt{)\{}c\texttt{\}else\{}c'\texttt{\}} \mid \ell\texttt{=}v.m\texttt{<}\bar{\pi}\texttt{>(}\bar{v}\texttt{)} \mid \texttt{assert(}F\texttt{)}$

Synt. Restr.: Logic variables that occur in $\ell\texttt{=new}\ C\texttt{<}\bar{\pi}\texttt{>}$ must be bound by class parameters.

Our command language assumes that Java-like commands have been transformed so that intermediate values are always assigned to local variables. Following [17], we assume that methods only return at the end of their body. We omit the `return`-command. Values are included in the syntax domain of commands, so that a terminating, non-blocking execution of a command results in the return value. Methods of type `void` return `null`, which is the only member of type `void`. We usually omit terminating occurrences of `null`. The operator for *existential unpacking* has no effect at runtime. It makes the existential variable α available in the continuation c for instantiation of logic method parameters. In examples, we often omit explicit existential unpacking and instantiation of logic method parameters. Making these explicit helps with the theory.

[6] In reality, they would be implemented natively.

2.2 Operational Semantics

Runtime Structures:

$\mathsf{ClVal} = \mathsf{Val} \setminus \mathsf{RdVar}$	closed values
$s \in \mathsf{Stack} = \mathsf{RdWrVar} \rightharpoonup \mathsf{ClVal}$	stacks
$t \in \mathsf{Thread} = \mathsf{Stack} \times \mathsf{Cmd} \ ::= \ s \ \mathsf{in} \ c$	threads
$ts \in \mathsf{ThreadPool} = \mathsf{ObjId} \rightharpoonup \mathsf{Thread} \ ::= \ o_1 \ \mathsf{is} \ t_1 \mid \cdots \mid o_n \ \mathsf{is} \ t_n$	thread pools
$os \in \mathsf{ObjStore} = \mathsf{FieldId} \rightharpoonup \mathsf{ClVal}$	object stores
$obj \in \mathsf{Obj} = \mathsf{Ty} \times \mathsf{ObjStore} \ ::= \ (T, os)$	objects
$h \in \mathsf{Heap} = \mathsf{ObjId} \rightharpoonup \mathsf{Obj}$	heaps
$st \in \mathsf{State} = \mathsf{Heap} \times \mathsf{ThreadPool} \ ::= \ \langle h, ts \rangle$	states
$prog \in \mathsf{Program} = \mathsf{ClassTable} \times \mathsf{Cmd} \ ::= \ (ct, c)$	programs

Each thread "s in c" consists of a thread-local stack s and a process continuation c. In thread pools, each thread t is associated with a unique object identifier, which serves as a thread identifier. The dynamic semantics of our language is a small-step operational semantics $st \rightarrow_{ct} st'$ and can be found in [16].

There is one (and only one) reduction rule where our operational semantics depends on class parameters, namely the reduction rule for type casts. Downcasts to parametrized types require a runtime check that looks at the type parameters, which the standard JVM does not keep track of. There are at least three ways how one could deal with that in practice: Firstly (and most pragmatically), one could simply forbid downcasts to reference types that have a non-empty parameter list. Secondly, one could develop an enhanced virtual machine that keeps track of class parameters. Thirdly, one could devise a syntactic translation that erases class parameters such that the target of this translation throws a `ClassCastException` whenever the source does.

2.3 Specification Formulas and Their Semantics

Specification Formulas:

$e \in \mathsf{Exp} \ ::= \ \pi \mid \ell \mid op(\bar{e}) \qquad lop \in \{\texttt{*}, \texttt{-*}, \texttt{\&}, \texttt{|}\} \qquad qt \in \{\texttt{ex}, \texttt{fa}\}$

$\kappa \in \mathsf{Pred} ::=$	predicates
P	P at receiver's dynamic class
$P\texttt{@}C$	P at class C
$E, F, G, H \in \mathsf{Formula} ::=$	specification formulas
e	boolean expression
$\texttt{PointsTo}(e.f, \pi, e')$	$e.f$ points to e' and the access permission for $e.f$ is π
$\texttt{Perm}(e.\texttt{join}, \pi)$	permission to use a split of `join`'s postcondition
$\pi.\kappa\texttt{<}\bar{\pi}'\texttt{>}$	predicate $\pi.\kappa$ applied to $\bar{\pi}'$
$F \ lop \ G$	binary logical operator
$(qt \ T \ \alpha)(F)$	quantifier

Derived forms: $F \texttt{ *-* } G \stackrel{\Delta}{=} (F \texttt{ -* } G) \texttt{ \& } (G \texttt{ -* } F) \qquad F \texttt{ assures } G \stackrel{\Delta}{=} F \texttt{ -* } (F \texttt{ * } G)$

$F \texttt{ ispartof } G \stackrel{\Delta}{=} G \texttt{ -* } (F \texttt{ * } (F \texttt{ -* } G))$

The formula semantics is defined by a Kripke resource interpretation [23] of the form $\Gamma \vdash \mathscr{E}; \mathscr{R}; s \models F$, where Γ is a type environment, $\mathscr{E}$ is a predicate environment that maps predicate names to predicates, $\mathscr{R}$ is a resource, and s is a stack. Resources are triples $\mathscr{R} = (h, \mathscr{P}, \mathscr{Q})$ of heaps h and two permission tables $\mathscr{P}$ and $\mathscr{Q}$. Permission

tables are functions of type $\mathsf{ObjId} \times (\mathsf{FieldId} \times \{\texttt{join}\}) \to [0,1]$ that map fields and join to fractional permissions. The resource components h and $\mathscr{P}$ are local resources, whereas $\mathscr{Q}$ is a global resource. We denote the projections to the resource components by $\mathscr{R}_{\mathsf{hp}}$, $\mathscr{R}_{\mathsf{loc}}$ and $\mathscr{R}_{\mathsf{glo}}$. The definition of the forcing relation $\models$ is pretty standard, and we refer to [16] for details.

2.4 Soundness Theorems

Below, we define a verification system whose top level judgment is $prog : \diamond$ (read: "*prog* is verified"). We have proven a *preservation theorem* from which we can draw several corollaries, namely, *data race freedom*, *null error freedom* and *partial correctness*.

A pair (hc, hc') of head commands is called a *data race* iff $hc = (o.f\,\texttt{=}\,v)$ and either $hc' = (o.f\,\texttt{=}\,v')$ or $hc' = (\ell\,\texttt{=}\,o.f)$ for some o, f, v, v', ℓ. A head command hc is called a *null error* iff $hc = (\ell\,\texttt{=}\,\texttt{null}.f)$ or $hc = (\texttt{null}.f\,\texttt{=}\,v)$ or $hc = (\ell\,\texttt{=}\,\texttt{null}.m\texttt{<}\bar{\pi}\texttt{>}(\bar{v}))$. We define *initial states*: $\mathsf{init}(c) = \langle \{\mathsf{main} \mapsto (\texttt{Thread}, \emptyset)\}, \mathsf{main}\ \mathsf{is}\ (\emptyset\ \mathsf{in}\ c) \rangle$, where main is some distinguished object id 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 $\emptyset$).

Theorem 1 (Verified Programs are Data Race Free). *If* $(ct, c) : \diamond$ *and* $\mathsf{init}(c) \to_{ct}^{*} \langle h, ts \mid o_1\ \mathsf{is}\ (s_1\ \mathsf{in}\ hc_1; c_1) \mid o_2\ \mathsf{is}\ (s_2\ \mathsf{in}\ hc_2; c_2) \rangle$*, then* (hc_1, hc_2) *is not a data race.*

Theorem 2 (Verified Programs are Null Error Free). *If* $(ct, c) : \diamond$ *and* $\mathsf{init}(c) \to_{ct}^{*} \langle h, ts \mid o\ \mathsf{is}\ (s\ \mathsf{in}\ hc; c) \rangle$*, then hc is not a null error.*

Theorem 3 (Partial Correctness).
If $(ct, c) : \diamond$ *and* $\mathsf{init}(c) \to_{ct}^{*} \langle h, ts \mid o\ \mathsf{is}\ (s\ \mathsf{in}\ \texttt{assert}(F); c) \rangle$*, then* $(\Gamma \vdash \mathscr{E}; \mathscr{R}; s \models F[\sigma])$ *for some* Γ, $\mathscr{E}$, $\mathscr{R}$ *such that* $\mathscr{R}_{\mathsf{hp}} = h$*, and* $\sigma \in \mathsf{LogVar} \rightharpoonup \mathsf{SpecVal}$.

3 The Verification System

3.1 Proof Theory

Many presentations of separation logic are based on a model-theoretic logical consequence. We, instead, define logical consequence proof-theoretically. This gives our system an algorithmic flavour, similar to recent static assertion checkers for fragments of separation logic [2,9] that are built upon proof-theoretic decision procedures[7].

$\Gamma; v; \bar{F} \vdash G$	from v's point of view, G is a logical consequence of the $\ast$-conjunction of $\bar{F}$
$\Gamma; v \vdash F$	from v's point of view, F is an axiom

In the former judgment, $\bar{F}$ is a *multiset* of formulas. The parameter v represents the *current receiver*, which is needed to determine the scope of predicate definitions.

The logical consequence judgment is driven by standard natural deduction rules that are common to the logic of bunched implications [23] and linear logic [30]. These rules are detailed in [16]. We admit weakening, because Java is a garbage-collected language. The link between $\Gamma; v; \bar{F} \vdash G$ and the axiom judgment $\Gamma; v \vdash F$ is established by the following rule. (We omit the definitions of typing judgments $\Gamma \vdash v : T$ and $\Gamma \vdash F : \diamond$.)

[7] Unfortunately, these fragments do not include $-\!\ast$, as needed for our iterator implementation.

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

We now define the complete set of axioms. First, we repeat the split/merge law:

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

$$\Gamma;v \vdash \texttt{Perm}(e.\texttt{join},\pi) \texttt{ *-* } (\texttt{Perm}(e.\texttt{join},\tfrac{\pi}{2}) \texttt{ * } \texttt{Perm}(e.\texttt{join},\tfrac{\pi}{2}))$$

For the following axioms, recall that "F `assures` G" abbreviates "F `-*` $(F * G)$".

$$\Gamma;v \vdash \texttt{true} \qquad \Gamma;v \vdash \texttt{false -* } F \qquad \Gamma;v \vdash (e \,\&\, F) \texttt{ -* } (e * F)$$

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

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

The third of these axioms implies that boolean expressions are copyable: e `-*` $(e * e)$.

The following axiom lifts semantic validity of boolean expressions (which we do not axiomatize) to our proof theory:

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

The next axiom allows to apply class axioms. Here, $\mathsf{axiom}(t\texttt{<}\bar{\pi}'\texttt{>})$ is the $*$-conjunction of all class axioms in $t\texttt{<}\bar{\pi}'\texttt{>}$ and its supertypes.

$$(\Gamma \vdash \pi : t\texttt{<}\bar{\pi}'\texttt{>} \;\wedge\; \mathsf{axiom}(t\texttt{<}\bar{\pi}'\texttt{>}) = F) \;\Rightarrow\; \Gamma;v \vdash F[\pi/\texttt{this}]$$

The *open/close axiom* allows predicate receivers to replace abstract predicates by their definitions. It uses a function $\mathsf{pbody}(v.P\texttt{<}\bar{\pi}\texttt{>}, C\texttt{<}\bar{\pi}'\texttt{>})$ that returns the extension F of predicate $v.P\texttt{<}\bar{\pi}\texttt{>}$ in class $C\texttt{<}\bar{\pi}'\texttt{>}$.

$$\begin{array}{l}(\Gamma \vdash v : C\texttt{<}\bar{\pi}''\texttt{>} \;\wedge\; \mathsf{pbody}(v.P\texttt{<}\bar{\pi},\bar{\pi}'\texttt{>}, C\texttt{<}\bar{\pi}''\texttt{>}) = F \;\wedge\; C \prec_1 D) \\ \quad \Rightarrow\; \Gamma;v \vdash v.P\texttt{@}C\texttt{<}\bar{\pi},\bar{\pi}'\texttt{>} \texttt{ *-* } (F * v.P\texttt{@}D\texttt{<}\bar{\pi}\texttt{>})\end{array}$$

Note that the current receiver, as represented on the left of $\vdash$, 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 a higher arity than $P\texttt{@}D$: following [26] we allow subclasses to extend predicate arities.

The following axiom deals with unqualified predicates with missing parameters:

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

The following axioms capture additional facts about abstract predicates. Recall that "F `ispartof` G" is defined as G `-*` $(F * (F \texttt{ -* } G))$.

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

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

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

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

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

Here, the expression "π `isclassof` C" evaluates to `true` whenever C is π's dynamic class. "C `isclassof` π" surely holds right after object π of class C has been created. Consequently, our Hoare rules introduce it as a postcondition of object creation commands. The second axiom makes use of `final` classes (resp. predicates), which are classes (resp. predicates) that are prohibited to be extended.

3.2 Method Subtyping

Method types are of the following form:

$$\texttt{<}\bar{T}\,\bar{\alpha}\texttt{>}\,\texttt{req}\,F;\,\texttt{ens}\,G;\,U\,m\texttt{(}V_0\,\iota_0;\bar{V}\,\bar{\iota}\texttt{)}$$

In method types, we make the self-parameter explicit, separated from the other formal parameters by a semicolon. In the scheme above, ι_0 is the self-parameter.

Before presenting the method subtyping rule in full generality, we present its instance for method types without logic parameters:

$$\frac{U,V_0,\bar{V}' <: U',V_0',\bar{V} \quad \Gamma,\iota_0:V_0,\bar{\iota}:\bar{V}';\ \iota_0;\ \texttt{true} \vdash F' \mathbin{-\!*} (F * \texttt{(fa}\,U\,\texttt{result)}(G \mathbin{-\!*} G'))}{\Gamma \vdash \texttt{req}\,F;\,\texttt{ens}\,G;\,U\,m\texttt{(}V_0\,\iota_0;\bar{V}\,\bar{\iota}\texttt{)} <: \texttt{req}\,F';\,\texttt{ens}\,G';\,U'\,m\texttt{(}V_0'\,\iota_0;\bar{V}'\,\bar{\iota}\texttt{)}}$$

This rule has the following two derived rules (where types are elided):

$$\frac{\vdash F' \mathbin{-\!*} F \quad \vdash G' \mathbin{-\!*} G}{\vdash \texttt{req}\,F;\,\texttt{ens}\,G <: \texttt{req}\,F';\,\texttt{ens}\,G'} \qquad \frac{}{\vdash \texttt{req}\,F;\,\texttt{ens}\,G <: \texttt{req}\,F * H;\,\texttt{ens}\,G * H}$$

The first of these derived rules is standard behavioural 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):

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

The general method subtyping rule also accounts for logic parameters:[8]

$$\frac{\begin{array}{c} m \neq \texttt{run} \qquad \bar{T}',U,V_0,\bar{V}' <: \bar{T},U',V_0',\bar{V} \\ \Gamma,\iota_0:V_0;\iota_0;\texttt{true} \vdash \texttt{(fa}\,\bar{T}'\,\bar{\alpha}\texttt{)(fa}\,\bar{V}'\,\bar{\iota}\texttt{)}(F' \mathbin{-\!*} \texttt{(ex}\,\bar{W}\,\bar{\alpha}'\texttt{)}(F * \texttt{(fa}\,U\,\texttt{result)}(G \mathbin{-\!*} G'))) \end{array}}{\Gamma \vdash \texttt{<}\bar{T}\,\bar{\alpha},\bar{W}\,\bar{\alpha}'\texttt{>}\texttt{req}\,F;\,\texttt{ens}\,G;\,U\,m\texttt{(}V_0\,\iota_0;\bar{V}\,\bar{\iota}\texttt{)} <: \texttt{<}\bar{T}'\,\bar{\alpha}\texttt{>}\texttt{req}\,F';\,\texttt{ens}\,G';\,U'\,m\texttt{(}V_0'\,\iota_0;\bar{V}'\,\bar{\iota}\texttt{)}}$$

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

$$\frac{}{\vdash \texttt{<}T\,\alpha\texttt{>}\texttt{req}\,F;\,\texttt{ens}\,G <: \texttt{req}\,\texttt{(ex}\,T\,\alpha\texttt{)}(F);\,\texttt{ens}\,\texttt{(ex}\,T\,\alpha\texttt{)}(G)}$$

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

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

3.3 Hoare Triples

Our Hoare rules are syntax-directed, omitting structural rules. Separation logic's frame rule is admissible. Separation logic's auxiliary variable rule is subsumed by our syntax for existential unpacking. We omit the rules of conjunction and disjunction, and did not need them in the examples we considered. We could soundly add the rule of disjunction. To add the rule of conjunction, we would need to assume that preconditions of `run()` are *supported* [14].[9]

[8] The subtyping rule for `run` is restricted to avoid dependencies between pre- and postcondition.

[9] Supported formulas are formulas that have the property that, for any resource, the set of subresources that satisfy it is either empty or has a least element. They play a similar role for intuitionistic predicates, as *precise* formulas for non-intuitionistic predicates [24]. In our variant of separation logic, all predicates are intuitionistic, as we admit weakening.

Hoare triples have the forms $(\Gamma; v \vdash \{F\} c : T\{G\})$ and $(\Gamma; v \vdash \{F\} hc \{G\})$, where v is the receiver parameter. We present a few selected rules and refer to [16] for the complete rule system.

The rules for reading and writing fields are standard:

$$\frac{\Gamma; v; F \vdash \texttt{PointsTo}(w.f, \pi, u) \quad \Gamma \vdash w : U \quad T\, f \in \mathsf{fld}(U) \quad T[w/\texttt{this}] <: \Gamma(\ell) \quad \ell \notin F}{\Gamma; v \vdash \{F\} \ell \texttt{=} w.f \{F * \ell \texttt{==} u\}}$$

$$\frac{\Gamma \vdash v, F : \texttt{Object}, \diamond \quad \Gamma \vdash u : U \quad T\, f \in \mathsf{fld}(U) \quad \Gamma \vdash w : T[u/\texttt{this}]}{\Gamma; v \vdash \{F * \texttt{PointsTo}(u.f, 1, T)\} u.f \texttt{=} w \{F * \texttt{PointsTo}(u.f, 1, w)\}}$$

The rule for forking a thread consumes `run`'s precondition. The postcondition of `fork()` is empty.[10] The rule makes use of the function $\mathsf{mtype}(m, T)$, which looks up m's type in the smallest supertype of T that declares m:

$$\frac{\begin{array}{c}\mathsf{mtype}(\texttt{run}, T) = \texttt{req}\, G; \texttt{ens}\, G'; \texttt{void run(}T\, \iota_0\texttt{;)} \\ \ell \notin F \quad \Gamma(\ell) = \texttt{void} \quad \Gamma \vdash u : T <: \texttt{Thread} \quad \Gamma; v; F \vdash u \texttt{!=null}\end{array}}{\Gamma; v \vdash \{F * G[u/\iota_0]\} \ell \texttt{=} u.\texttt{fork()} \{F\}}$$

The most interesting rule is the one for joining threads. It allows the caller to exchange a fraction *fr* of the `join`-permission $\texttt{Perm}(u.\texttt{join}, 1)$ for a fraction *fr* of u.`run`'s postcondition:[11]

$$\frac{\begin{array}{c}\mathsf{mtype}(\texttt{run}, T) = \texttt{req}\, G; \texttt{ens}\, G'; \texttt{void run(}T\, \iota_0\texttt{;)} \quad fr = \mathsf{all} \text{ or } G' \text{ is supported} \\ \ell \notin F \quad \Gamma(\ell) = \texttt{void} \quad \Gamma \vdash u : T <: \texttt{Thread} \quad \Gamma; v; F \vdash u \texttt{!=null}\end{array}}{\Gamma; v \vdash \{F * fr \cdot \texttt{Perm}(u.\texttt{join}, 1)\} \ell \texttt{=} u.\texttt{join()} \{F * fr \cdot G'[u/\iota_0]\}}$$

Here, *fr* ranges over *linear combinations*. These represent numbers of the forms 1 or $\sum_{i=1}^{n} bit_i \cdot \frac{1}{2^i}$:

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

To define the scalar multiplication $fr \cdot F$, we first extend the split-operation from permissions to formulas:

$$\begin{array}{c}\mathsf{split}(e) \triangleq e \qquad \mathsf{split}(\pi.\kappa\texttt{<}\bar{\pi}'\texttt{>}) \triangleq \pi.\kappa\texttt{<}\mathsf{split}(\bar{\pi}')\texttt{>} \\ \mathsf{split}(\texttt{PointsTo}(e.f, \pi, e')) \triangleq \texttt{PointsTo}(e.f, \texttt{split}(\pi), e') \\ \mathsf{split}(\texttt{Perm}(e.\texttt{join}, \pi)) \triangleq \texttt{Perm}(e.\texttt{join}, \texttt{split}(\pi)) \\ \mathsf{split}(F\ lop\ G) \triangleq \mathsf{split}(F)\ lop\ \mathsf{split}(G) \quad \mathsf{split}(\texttt{(}qt\ T\ \alpha\texttt{)}(F)) \triangleq \texttt{(}qt\ T\ \alpha\texttt{)}(\mathsf{split}(F))\end{array}$$

Now, the scalar multiplication $fr \cdot F$ is defined as follows: $\mathsf{all} \cdot F = F$, $\mathsf{fr}() \cdot F = \texttt{true}$, $\mathsf{fr}(1) \cdot F = \mathsf{split}(F)$, $\mathsf{fr}(0, bits) \cdot F = \mathsf{fr}(bits) \cdot \mathsf{split}(F)$, and $\mathsf{fr}(1, bits) \cdot F = \mathsf{split}(F) * \mathsf{fr}(bits) \cdot \mathsf{split}(F)$. For instance, $\mathsf{fr}(1,0,1) \cdot F \texttt{ *-* } (\mathsf{split}(F) * \mathsf{split}^3(F))$.

Via the bijection $\mathsf{fr}(bits) \mapsto \sum_{i=1}^{n} bit_i \cdot \frac{1}{2^i}$, we can define an addition on linear combinations that reflects the addition on concrete binary fractions. For proving soundness of the `join()`-rule, it is crucial that `join()`'s postcondition satisfies the following distributivity law, which holds if G' is supported:

$$(fr_1 + fr_2) \cdot G' \texttt{ *-* } (fr_1 \cdot G' * fr_2 \cdot G')$$

[10] The permission $\texttt{Perm}(u.\texttt{join}, 1)$ gets introduced when the thread object u is created.

[11] We assume that postconditions of methods with return type `void` do not mention the `result`-variable.

4 Comparison to Related Work and Conclusion

Parkinson/Bierman are the first to adapt separation logic to a Java-like language [25,26]. We build on their work, using abstract predicates, but extend it to a concurrent language and combine abstract predicates with fractional permissions.

Boyland and Retert [7] explain the relation between write-effects, uniqueness and datagroups in terms of a linear type-and-effect system. Their system features a nesting operation and recursive definitions, which serve as an abstraction mechanism similar to abstract predicates, but in addition promote linear formulas to non-linear ones. Recently, Boyland presented a semantics for formulas that combine nesting and fractional permissions [6]. His semantics is quite different from ours. Generally speaking, our semantics is closer to standard semantics of BI [23]. Boyland facilitates permission splitting for datagroups through an operation that scales formulas by fractions, whereas we require datagroups to be fully permission-parametrized and scale the parameters. Because we allow multiple parameters, our approach permits more fine-grained scaling for overlapping datagroups (see Section 1.6 for an example).

Bierhoff and Aldrich [3] combine typestates and fractional permissions to specify object usage protocols. They use iterators as an example, but they do not allow linear implications in method contracts. As a result, their usage protocol regulates access to the collection itself, but not access to the elements of the collection, and their protocol would not prevent data races in concurrent programs. Krishnaswami [19] (in higher-order separation logic) and Boyland et al [8] (in their linear type-and-effect system) present iterator contracts that use linear implication and are related to ours.

Gotsman et al [14] recently adapted concurrent separation logic to Posix threads, treating storable locks. They do not support read-sharing of join's postcondition like us.

Regarding the interplay between abstract predicates and subclassing, we axiomatize the "stack of class frames" [12,1] to control predicate extensions in subclasses. The stack of class frames supports the use of subclassing for specialization and is well-suited for dealing with extended object state. Furthermore, the stack of class frames facilitates fully modular verification, avoiding the need to re-verify inherited methods, which is required in [25,26] where unrestricted predicate re-definitions in subclasses are allowed. In recent work, Parkinson and Bierman argue that a verification systems should support subclassing for code reuse in addition to subclassing for specialization, and present a system that supports both uses of subclassing while avoiding re-verification of inherited methods [27]. To this end, they associate with each method *two* contracts: a concrete "static" contract, and an abstract "dynamic" contract. Their system checks that predicate re-definitions in subclasses are compatible with concrete static contracts of inherited methods, thereby avoiding re-verification of implementations of inherited methods. The advantage over the stack of class frames is increased flexibility, the disadvantage is heavier specification machinery, although much of this can be hidden behind good defaults. Chin et al [9] make a similar proposal.

Conclusion. We have presented a variant of concurrent separation logic with fractional permissions for a Java-like language with fork/join and proved it sound. Future work includes algorithmic checking and extension to handle lock synchronization.

Acknowledgments. We thank John Boyland, Marieke Huisman, Erik Poll and anonymous reviewers for their very useful comments that helped to improve this paper.

References

1. 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.
2. J. Berdine, C. Calcagno, P. W. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *Formal Methods for Components and Objects*, 2005.
3. K. Bierhoff, J. Aldrich. Modular typestate verification of aliased objects. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, 2007.
4. R. Bornat, P. O'Hearn, C. Calcagno, M. Parkinson. Permission accounting in separation logic. In *Principles of Programming Languages*, New York, NY, USA, 2005. ACM Press.
5. 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.
6. J. Boyland. Semantics of fractional permissions with nesting. Technical report, University of Wisconsin at Milwaukee, 2007.
7. J. Boyland, W. Retert. Connecting effects and uniqueness with adoption. In *Principles of Programming Languages*, 2005.
8. J. Boyland, W. Retert, Y. Zhao. Iterators can be independent "from" their collections. International Workshop on Aliasing, Confinement and Ownership in object-oriented programming, 2007.
9. W. Chin, C. David, H. Nguyen, S. Qin. Enhancing modular OO verification with separation logic. In *Principles of Programming Languages*, 2008.
10. K. Crary, D. Walker, G. Morrisett. Typed memory management in a calculus of capabilities. In *Principles of Programming Languages*, 1999.
11. R. DeLine, M. Fähndrich. Enforcing high-level protocols in low-level software. In *Programming Languages Design and Implementation*, 2001.
12. R. DeLine, M. Fähndrich. Typestates for objects. In *European Conference on Object-Oriented Programming*, 2004.
13. J.-Y. Girard. Linear logic: Its syntax and semantics. In J.-Y. Girard, Y. Lafont, L. Regnier, eds., *Advances in Linear Logic*. Cambridge University Press, 1995.
14. A. Gotsman, J. Berdine, B. Cook, N. Rinetzky, M. Sagiv. Local reasoning for storable locks and threads. In *Asian Programming Languages and Systems Symposium*, 2007.
15. C. Haack, C. Hurlin. Resource usage protocols for iterators. http://www.cs.ru.nl/~chaack/papers/iterators.pdf.
16. C. Haack, C. Hurlin. Separation logic contracts for a Java-like language with fork/join. Technical Report 6430, INRIA, 2008.
17. A. Igarashi, B. Pierce, P. Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Trans. Program. Lang. Syst.*, 23(3), 2001.
18. S. Ishtiaq, P. O'Hearn. BI as an assertion language for mutable data structures. In *Principles of Programming Languages*, 2001.
19. G. Krishnaswami. Reasoning about iterators with separation logic. In *Specification and Verification of Component-Based Systems*, 2006.
20. G. T. Leavens, A. L. Baker, C. Ruby. Preliminary design of JML: a behavioral interface specification language for Java. *SIGSOFT Software Engineering Notes*, 31(3), 2006.
21. K. R. M. Leino. Data groups: Specifying the modification of extended state. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, 1998.
22. P. O'Hearn. Resources, concurrency and local reasoning. *Theor. Comp. Science*, 375(1–3), 2007.
23. P. W. O'Hearn, D. J. Pym. The logic of bunched implications. *Bulletin of Symbolic Logic*, 5(2), 1999.
24. P. W. O'Hearn, H. Yang, J. C. Reynolds. Separation and information hiding. In *Principles of Programming Languages*, Venice, Italy, 2004. ACM Press.
25. M. Parkinson. Local reasoning for Java. Technical Report UCAM-CL-TR-654, University of Cambridge, 2005.
26. M. Parkinson, G. Bierman. Separation logic and abstraction. In *Principles of Programming Languages*, 2005.
27. M. Parkinson, G. Bierman. Separation logic, abstraction and inheritance. In *Principles of Programming Languages*, 2008.
28. J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Logic in Computer Science*, Copenhagen, Denmark, 2002. IEEE Press.
29. 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.
30. P. Wadler. A taste of linear logic. In *Mathematical Foundations of Computer Science*, 1993.