JOURNAL OF OBJECT TECHNOLOGY
Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering. ©JOT, 2008

Vol. 8, No. 4, Special Issue: Workshop on FTfJP and IWACO at ECOOP 08, June 2009

Resource Usage Protocols for Iterators

Christian Haack[1], Radboud University Nijmegen, The Netherlands
Clément Hurlin[12], INRIA Sophia Antipolis – Méditerranée, France and University of Twente, The Netherlands

We discuss usage protocols for iterator objects that prevent concurrent modifications of the underlying collection while iterators are in progress. We formalize these protocols in Java-like object interfaces, enriched with separation logic contracts. We present examples of iterator clients and proofs that they adhere to the iterator protocol, as well as examples of iterator implementations and proofs that they implement the iterator interface.

1 Introduction

Objects are often meant to be used according to certain protocols. In many cases, such protocols impose temporal constraints on the order of method calls. A simple example are protocols for output streams that impose that clients do not write to streams after the streams have been closed. Whereas object interfaces in typed object-oriented languages formally specify type signatures, they typically do not formalize such object usage protocols. To improve on this, researchers have recently spent considerable efforts on designing formal specification and verification systems for object usage protocols. Some of these systems are based on classical program logics, using ghost variables [PBB$^+$04] and temporal logic extensions [TH02]. The problem with these specification techniques is that static verification of protocol adherence is difficult because of aliasing. So-called *typestate systems* have focused on automatic static checkability. In order to deal with the aliasing problem, these systems employ linear type-and-effect systems [DF01, DF04] and ideas from linear logic [BA07].

While in simple cases usage protocols only constrain method call order, more sophisticated objects need more intricate temporal constraints. A prominent example are iterator objects, as featured in languages like Java or C♯. An iterator usage protocol is meant to ensure that iterator calls for retrieving the next collection element always successfully return a new element. Furthermore, an iterator call to remove the last retrieved element should indeed remove this element, and not perhaps an element that has been retrieved much earlier in the iteration. In order to make such guarantees, usage protocols for iterators must prevent so-called *concurrent modifications* of the underlying collection: while an iterator is in progress, the collection should not get modified by other actions that interleave with

[0] This is an extended version of a paper at the International Workshop on Aliasing, Ownership and Confinement (IWACO 2008).
[1] Supported in part by IST-FET-2005-015905 Mobius project.
[2] Supported in part by ANR-06-SETIN-010 ParSec project.

Christian Haack, Clément Hurlin: "Resource Usage Protocols for Iterators", in *Journal of Object Technology*, vol. 8, no. 4, Special Issue: Workshop on FTfJP and IWACO at ECOOP 08, June 2009, pages 55–83, http://www.jot.fm/issues/issue_2009_06/article3/

```
interface Collection {
    Iterator iterator();
}

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

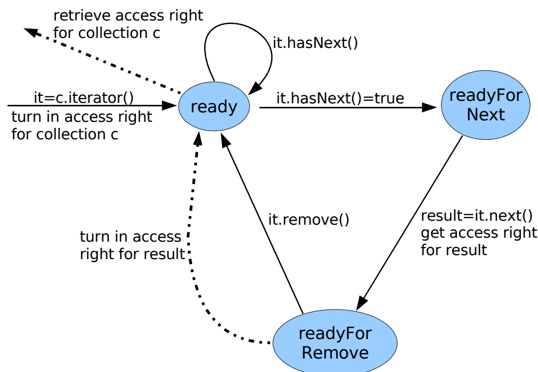


Figure 1: Basic iterator protocol

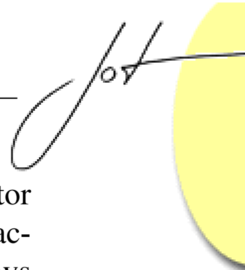
the iteration[1]. If concurrent modifications were not prevented, then the above mentioned guarantees could get violated by concurrently removing collection elements (in the extreme case, concurrently clearing the collection entirely). Thus, an iterator usage protocol must prevent concurrent modifications.

There is another reason why it is desirable to constrain concurrent collection access through iterators: often iterators temporarily break collection invariants. Consider, for instance, a list of point objects that represents a polygon. Such a list satisfies the invariant that any two lines connecting adjacent points do not cross. It is likely that an iterator over the point list that moves the polygon breaks this invariant temporarily: lines connecting points that have already been moved may cross lines connecting points that have not yet been moved. In such inconsistent intermediate states, methods that assume a consistent collection should not access the collection. The iterator protocols presented in this paper specify such dynamic access policies abstractly.

Figure 1 represents a usage protocol for iterators as a state machine. The protocol prevents concurrent modifications and runtime exceptions due to iteration beyond the end of the collection. We now explain the protocol: Following the permission interpretation of separation logic, each piece of heap space is associated with an unforgeable permission to access this space[2]. Such permissions are abstract entities; they are not represented or checked at runtime, and are only used in static verification rules. According to our protocol, when an iterator over collection `c` gets created, the caller of `c.iterator()` temporarily abandons the access permission for `c`. Iteration is then governed by a three-state protocol. The solid state transitions in the picture are associated with method calls. For instance, in the `ready`-state the method `it.hasNext()` can be called an arbitrary number of times. When `it.hasNext()` returns `true`, the iterator client has the option to either move to the `readyForNext`-state or stay in the `ready`-state. Once in the `readyForNext`-state, the iterator client may call `it.next()`. Note that the protocol enforces that `it.next()` can only be called after `it.hasNext()` has returned `true` at least once. This policy prevents runtime exceptions due to iterations beyond the end of

[1] Such interleaving actions may execute in the same thread as the iterator.

[2] We use the words *permission*, *access right*, *access ticket* interchangeably.



the collection (`NoSuchElementException` in Java). Calling `it.next()` takes the iterator client into the `readyForRemove`-state, and furthermore gives the client permission to access the space that is associated with the returned collection element. There are two ways to go on from the `readyForRemove`-state: either remove the previously returned element from the collection by calling `it.remove()` (in this case the access right for the removed element stays with the client), or abandon the access right for the previously returned element. The latter state transition is not associated with a method call or any other concrete runtime event, and for that reason we have represented it in the picture by a dashed arrow. At runtime, this dashed state transition "happens" somewhere between the last access to the state of the previously returned collection element, and the first concrete event that is enabled in the `ready`-state or in a state that can be reached from the `ready`-state by a sequence of dashed transitions.

We now express this protocol as a contract in our specification language [HH08a], which is based on intuitionistic separation logic [IO01, Rey02, PB05]. Compared to standard presentations of separation logic, a peculiarity of [HH08a] is that we define logical consequence proof-theoretically, using a natural deduction calculus that is common to the (affine) logic of bunched implication (BI) [OP99] and (affine) linear logic [Gir87]. Our specification language has just one implication, namely -* (called *magic wand*, *separating implication* or *resource implication* in BI, and *linear implication* in linear logic). Compared to full BI, having just one implication has the advantage that it simplifies the natural deduction rules, because no bunched contexts are needed. The usual intuitive interpretations of the linear logic operators (as for instance explained in [Wad93], and as we explain below) are sound with respect to the Kripke resource semantics of separation logic. In particular, we can soundly represent state transitions by linear implications, as advocated by [Gir95] in Section 1.1.4. We find that the linear logic interpretation of the logical connectives very intuitively relates to the so-called "permission-reading" of separation logic.

Here is the `Iterator` interface that formalizes Figure 1:

```
interface Collection {
    //@ req this.space; ens result.ready;
    Iterator/*@<this>@*/ iterator();
}
interface Iterator/*@<Collection iteratee>@*/ {
    //@ pred ready;
    //@ pred readyForNext;
    //@ pred readyForRemove<Object element>;
    //@ axiom ready -* iteratee.space;
    //@ axiom (fa Object e)(readyForRemove<e> * e.space -* ready);
    //@ req ready; ens ready & (result -* readyForNext);
    boolean hasNext();
    //@ req readyForNext; ens readyForRemove<result> * result.space;
    Object next();
```

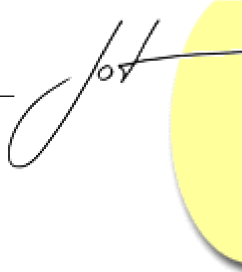
```
    //@ req readyForRemove<Object>; ens ready;
    void remove();
}
```

This interface declares three *heap predicates* `ready`, `readyForNext` and `readyForRemove`. Interface implementations must define these predicates in terms of concrete separation logic formulas. The predicate definitions must be such that the two *class axioms* are tautologically true, and that the methods satisfy their contracts (after replacing abstract predicate symbols in method contracts by concrete predicate definitions). In *method contracts*, the keyword `req` indicates the beginning of the *precondition* (aka *requires-clause*), and the keyword `ens` the beginning of the *postcondition* (aka *ensures-clause*). Each class extends a generic predicate `space`, which has a default definition in the `Object` class. This predicate should define the heap space that is associated with an object — often consisting of the object fields only, but in the case of collections sometimes also including the object spaces of the collection elements. Reference types and predicates may be parametrized by values. For instance, the `Iterator` class is parametrized by the collection, and the `readyForRemove` predicate is parametrized by the collection element that is ready to be removed. Parameters of types and predicates are logical variables, i.e., in contrast to program variables they are immutable. The *resource conjunction* $F * G$ expresses that both resources F and G are independently available: using either of these resources leaves the other one intact. The `&`-operator represents *choice*. If $F \& G$ holds, then F and G are available, but are interdependent: using either one of them destroys the other, too. The `&`-operator can represent non-deterministic state transitions. This is exhibited in the postcondition of `hasNext()`, which says that one can either stay in the `ready` state, or move on to the `readyForNext` state if the `result` of `hasNext()` was `true`. The *resource implication* $F \mathrel{-\!\!*} G$ grants the right to consume F yielding G in return. For instance, the first axiom in the `Iterator` interface says that one can always leave the `ready` state and in return get the full access permission on the underlying collection back. The second axiom says that one can always leave the state `readyForRemove<e>` and in return get into the `ready` state, provided one also abandons the permission to access the heap space associated with element `e`. Note how state transitions are directly represented by linear implications `-*`. Boolean expressions e are treated as *copyable* resources, i.e., they satisfy $e \mathrel{-\!\!*} (e * e)$. An example of a boolean expression is the `result`-variable in `hasNext()`'s postcondition[3].

The basic iterator protocol above has several shortcomings: it does not support multiple read-only iterators over the same collection, it does not support unrestricted access to immutable collection elements[4], and it does not support collections where the element access rights stay with the elements rather than being governed by the collection. In the

[3] For separation logic experts, we note that each `-*` in the iterator interface can equivalently be represented by $\Rightarrow$: in the postcondition of `hasNext()` because the antecedent is pure, and in the axioms because in intuitionistic separation logic `true` $\models F \Rightarrow G$ iff `true` $\models F \mathrel{-\!\!*} G$. In the definition of the `Iterator` predicates (see Figure 9 and the definition of `diff` on page 70), however, we use a true resource implication.

[4] We mean *persistent* immutability. By persistently immutable state, we mean state that stays immutable forever. In contrast, state that is associated with a fractional permission is temporarily immutable and may turn mutable again later.



remainder of this paper, we refine the basic protocol to address these shortcomings.

2 A Variant of Separation Logic for Java

We sketch our system from [HH08a, HHH08], which is based on intuitionistic separation logic.

We distinguish between *values* and *specification values*. The former include integers n, booleans b and read-only variables x. The latter, in addition, include fractional permissions [Boy03], which may occur in contracts and as type arguments, but not in executable code.

$n \in \mathsf{Int}$ *(integers)* $\quad b \in \mathsf{Bool}$ *(booleans)* $\quad x \in \mathsf{Var}$ *(logical variables)*
$v \in \mathsf{Val} ::= b \mid n \mid x$ *(values)*
$\pi \in \mathsf{SpecVal} ::= v \mid 1 \mid \mathtt{split}(\pi)$ *(specification values)*
Derived form: $\frac{\pi}{2^n} \stackrel{\Delta}{=} \mathtt{split}^n(\pi)$

Fractional permissions are binary fractions (i.e., fractions of the form $\frac{1}{2^n}$ where $n \geq 0$) in the interval $(0,1]$[5]. Fractional permissions have type `perm`. Predicates and types may be parametrized by fractional permissions. As usual [BOCP05], fractional permissions are arguments of the points-to predicate, in order to govern access rights : $v.f \stackrel{\pi}{\mapsto} e$ asserts that $v.f$ contains e and grants π-access to the field $v.f$. Writing the field requires 1-access, and reading it requires π-access for *some* π. The verification system ensures that, at each point in time, the sum of all fractional permissions for the same heap location is at most 1. As a result, the system prevents read-write and write-write conflicts, while permitting concurrent reads. The key for flexibly enforcing this global invariant is the *split-merge law*[6]:

$$v.f \stackrel{\pi}{\mapsto} e \;\; \text{*-*} \;\; (v.f \xrightarrow{\pi/2} e * v.f \xrightarrow{\pi/2} e)$$

Interfaces and classes can declare *predicates*. Semantically, these are predicates over heaps with at least one additional argument of type `Object` — the receiver. Predicates can be extended in subclasses in order to account for extended object state. Semantically, a predicate extension for predicate P defined in class C gets $*$-conjoined with the predicate extensions for P in C's superclasses.

$$P \in \mathsf{PredId} \qquad \kappa \in \mathsf{Pred} ::= P \mid P@C$$

The *qualified predicate* $v.P@C<\bar{\pi}>$ represents the conjunction of all predicate extensions for P in C's superclasses, up to and including C. The *unqualified predicate* $v.P<\bar{\pi}>$ is equivalent to $v.P@C<\bar{\pi}>$, where C is v's dynamic class. Our structured way of extending

[5]Most articles on fractional permissions use arbitrary fractions in the interval $(0,1]$. We use binary fractions and the split/merge law in order to avoid full rational arithmetic. Note that this way we can still split permissions into arbitrary numbers of permissions. For instance, we can split a 1-permission into three read-only permissions, namely one $\frac{1}{2}$-permission and two $\frac{1}{4}$-permissions.

[6] *-* is bi-directional resource implication, and defined as a derived form on the following page.

predicates facilitates modular verification (preventing re-verification of inherited methods), and is inspired by the so-called "stack of class frames" [DF04, BDF$^+$04].

Expressions are built from values and read-write variables, using a set of operators that includes standard relational and logical operators, and an operator C `isclassof` v that returns `true` iff C is v's dynamic class.

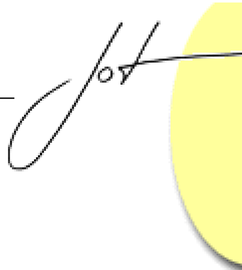
$$op \in \mathsf{Op} \supseteq \{\texttt{==}, \texttt{!=}, \texttt{!}, \texttt{\&}, \texttt{|}\} \cup \{C\ \texttt{isclassof} \mid C \in \mathsf{ClassId}\} \quad \textit{(boolean operators)}$$
$$\ell \in \mathsf{RdWrVar} \quad \textit{(stack variables)} \qquad e \in \mathsf{Exp} ::= \pi \mid \ell \mid op(\bar{e}) \quad \textit{(expressions)}$$

Formulas are built from boolean expressions, the points-to predicate and defined predicates, using a small set of logical operators.

$$lop \in \{\texttt{*}, \texttt{-*}, \texttt{\&}\} \quad \textit{(logical operators)} \qquad qt \in \{\texttt{ex}, \texttt{fa}\} \quad \textit{(quantifiers)}$$
$$F \in \mathsf{Formula} ::= e \mid v.f \overset{\pi}{\mapsto} e \mid v.\kappa\texttt{<}\bar{\pi}\texttt{>} \mid F\ lop\ F \mid (qt\ T\ x)(F) \quad \textit{(logical formulas)}$$
$$v.f \overset{\pi}{\mapsto} T \triangleq (\texttt{ex}\ T\ x)(v.f \overset{\pi}{\mapsto} x) \qquad F \texttt{*-*} G \triangleq (F \texttt{-*} G)\ \texttt{\&}\ (G \texttt{-*} F)$$
$$v.\kappa\texttt{<}\bar{\pi}, T, \bar{\pi}'\texttt{>} \triangleq (\texttt{ex}\ T\ x)(v.\kappa\texttt{<}\bar{\pi}, x, \bar{\pi}'\texttt{>}) \qquad F\ \texttt{ispartof}\ G \triangleq G \texttt{-*} (F \texttt{*} (F \texttt{-*} G))$$

Appendix A presents a typed variant of the standard natural deduction rules for (affine) linear logic (see, e.g., [Wad93] for a natural deduction presentation of linear logic). These are also the standard rules of the logic of bunched implication (BI) [OP99], as the natural deduction rules for linear logic and BI coincide for our restricted set of logical operators. Furthermore, the appendix presents axioms that capture specific properties of our particular model, namely typed heaps with subclassing and extensible abstract predicates. The appendix also presents Hoare rules for a small command language. These are standard separation logic rules, although we omit some structural rules that we do not need. The semantic interpretation of our formula language is standard and detailed in our technical report [HH08b]. There we also prove the soundness of the axioms with respect to the semantic model, as well as soundness of the Hoare rules: verified programs are partially correct, data-race free and never dereference `null`. For this paper (and perhaps for type-state protocols in general) the proof rules presented in the appendix are sufficient, although not complete with respect to the semantic model.

We assume that the `Object` class contains a default declaration of the `space`-predicate, as shown at the top of Figure 2. This predicate is meant to be extended in subclasses. Note that the axiom in `Object` imposes a constraint on the way subclasses may extend the `space`-predicate. In the axiom, we have omitted a leading universal quantifier over `p`. *By convention, free variables in class axioms that are not bound by class parameters are universally quantified in front of the axiom.* The axiom says that we can split/merge the `space`-predicate based on its permission parameter, just like points-to. This is not satisfied by all formulas (e.g., usually not satisfied by formulas that contain disjunctions or existentials with multiple witnesses), but is for instance satisfied by formulas of the shape $\texttt{space<p>} = \texttt{this}.f_1 \overset{\texttt{p}}{\mapsto} T_1 \texttt{ * } \cdots \texttt{ * this}.f_n \overset{\texttt{p}}{\mapsto} T_n$. In technical terms, sufficient criteria for predicates to satisfy a split/merge-axiom is that their defining formulas are "precise" or "supported" in the sense of [OYR04].



```
class Object {
    //@ pred space<perm p> = true;
    //@ axiom space<p> *-* (space<p/2> * space<p/2>);
}
final class MutableInteger {
    private int x;
    //@ pred space<perm p> = x ↦p int;
    //@ ens space<1>;
    public MutableInteger(int x) { this.x = x; }
    //@ <perm p> req space<p>; ens space<p>;
    public int get() { return x; }
    //@ req space<1>; ens space<1>;
    void set(int x) { this.x = x; }
}
final class Integer {
    private int x;
    //@ pred space<perm p> = (ex perm q)(x ↦q int);
    //@ axiom (fa perm p,q)(space<p> *-* space<q>);
    //@ <perm p> ens space<p>;
    public Integer(int x) { this.x = x; }
    //@ <perm p> req space<p>; ens space<p>;
    public int get() { return x; }
}
```

Figure 2: The `Object` class, and classes for mutable and immutable integer objects

Figure 2 also presents classes for mutable and immutable integer objects, which we will use in examples throughout this paper. In `MutableInteger`, we define the `space`-predicate as the points-to predicate to the object's `int`-field. The class axiom in `Object` trivially holds by the split/merge for points-to (see page 2 or axiom (Ax Split/Merge) on page 80). For immutable `Integer` objects, we encode immutability by existentially quantifying over a fractional permission — a standard encoding of immutability in terms of fractional permissions. Using the split/merge law and the natural deduction rules for existentials, one can show that `(ex perm q)(x` $\overset{q}{\longmapsto}$ `int)` is equivalent to `(ex perm q)(x` $\overset{q}{\longmapsto}$ `int) * (ex perm q)(x` $\overset{q}{\longmapsto}$ `int)`. As a result, the access permission for the heap space associated with immutable `Integer`s is copyable. This is captured by the class axiom in `Integer`, which allows to replace *v*.space<p> by *v*.space<q> for immutable `Integer`s *v* and any permission `p` and `q`. In method contracts, note that we explicitly quantify over auxiliary variables, enclosing quantifiers in angle brackets in front of method declarations (analogously to type parameters in Java).

3 Iterator Protocols

Protocol 1 — Permission-parametrized Iterator Type

Our first protocol parametrizes the `Iterator` interface by a fractional permission:

```
interface Collection {
    //@ req space<1> * e.space<1>; ens space<1>;
    void add(Object e);
```

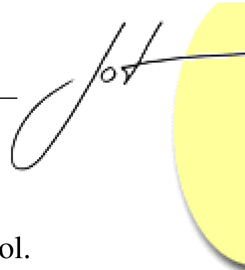
```
    //@ <perm p> req space<p>; ens result.ready;
    Iterator/*@<p,this>@*/ iterator();
}
interface Iterator/*@<perm p, Collection iteratee>@*/ {
    //@ pred ready;
    //@ pred readyForNext;
    //@ pred readyForRemove<Object element>;
    //@ axiom ready -* iteratee.space<p>;
    //@ axiom readyForRemove<e> * e.space<p> -* ready;
    //@ req ready; ens ready & (result -* readyForNext);
    boolean hasNext();
    //@ req readyForNext; ens readyForRemove<result> * result.space<p>;
    Object next();
    //@ req readyForRemove<Object> * p==1; ens ready;
    void remove();
}
```

To obtain the interface for a read-write iterator, one instantiates the permission parameter by 1. To obtain the interface for a read-only iterator, one instantiates the permission parameter by a fraction that is not known to be 1. Note that `remove()` can only be called on read-write iterators, because `p==1` is part of the precondition for `remove()`.

Figure 3 shows proof outlines for several clients of this interface (see the Hoare rules on page 80 for reference). The first client creates a read-write iterator `it` of type `Iterator<1,c>` over a collection `c` of mutable integer objects. The client then resets a collection element to 42 and removes another collection element. As a minor detail, note the application of (Ax Null) after the declaration of `y`. This axiom says that predicates with `null`-receivers (e.g., `null.space<1>`) are equivalent to `true`, by definition. In the example, the axiom applies because variable `y` is initialized with its default value `null`[7]. (Ax Null) turns out quite convenient in proofs. Intuitively, (Ax Null) is sound because no false facts can be derived from predicates with `null`-receivers. In particular, such predicates cannot be opened, because axiom (Ax Open/Close) only applies to predicates with non-`null` receivers (more specifically, to predicates whose receiver is `this`).

The second client in Figure 3 creates two read-only iterators `it1` and `it2` of type `Iterator<1/2,c>` over a single collection `c` of mutable integer objects. Neither `it1` nor `it2` can mutate or remove collection elements, but they can both retrieve and read elements. The third client creates an iterator `it` of type `Iterator<1,c>` over a collection of *immutable* integer objects. While the iteration is in progress, the immutable `Integer` object `zero` is accessed both through a direct reference to `zero` and through the iterator `it`. This is facilitated by the axiom in the `Integer` class, which is used to convert `zero.space<1/2>` to `zero.space<1>`. Note that for collections over *mutable* integer objects, the proof system forbids accessing collection elements directly while a read-write

[7]Our operational semantics (omitted in this paper) automatically initializes local variables with default values, which is convenient but not essential.



iterator is in progress.

We remark that the use of class axioms is not essential for expressing this protocol. Class axioms can be avoided by enriching the postconditions of `Collection.iterator()` and `Iterator.next()`. For instance, we could avoid the second class axiom if we "*-conjoined" this axiom with the postcondition of `next()`:

```
//@ req readyForNext;
//@ ens readyForRemove<result> * result.space<p>
//@    * (result.space<p> * readyForRemove<result> -* ready);
Object next();
```

Technically, the enriched postcondition is slightly weaker than the class axiom, because class axioms can be used arbitrarily often whereas postconditions can only be used once. However, this does not seem to restrict practical iterator clients because the class axiom is not meant to be used more often than `next()` is called.

Protocol 2 — Supporting Shallow Collections

In Protocol 1, access to mutable collection elements is governed by the collection. This may sometimes be inappropriate. Consider, for instance, a collection that represents a registry of mutable elements. Here, a good architecture may let the collection handle the adding and removing of elements, while leaving the element access rights with the client who registered the element.

For the sake of discussion, let's call collections that do not have access to their elements *shallow collections*, and collections that govern access to their elements *deep collections*. It is likely that most of the time even shallow collections need access to part of their elements' states—for instance the key in case of map entries. To safely share element state between a collection and its clients, the shared state (e.g., the key of a map entry) has to be immutable. We therefore introduce a second predicate `ispace` in the `Object` class to represent the immutable part of an object space. This is shown in Figure 4, together with a `MapEntry` class, where the `key`-field constitutes the immutable part and the `val`-field the mutable part. An axiom in the `Object` class formalizes that the immutable space is copyable.

In order to have a uniform type for deep and shallow collections, we parametrize the collection type by a boolean flag `isdeep`. When we instantiate this flag by `true`, we obtain a deep collection. Instantiating `isdeep` by `false` results in a shallow collection where the access rights to mutable parts of collection elements remain with the elements.

```
interface Collection/*@<boolean isdeep>@*/ {
    //@ req space<1> * e.ispace * (isdeep -* e.space<1>); ens space<1>;
    void add(Object e);
    //@ <perm p> req space<p>; ens result.ready;
    Iterator/*@<p,isdeep,this>@*/ iterator();
}
```

Read-write iterator:

```
Collection c = new List();
    { c.space<1> }
MutableInteger i0 = new MutableInteger(0);
    { i0.space<1> * c.space<1> }
c.add(i0);
    { c.space<1> }
MutableInteger i1 = new MutableInteger(1);
c.add(i1);
    { c.space<1> }
Iterator<1,c> it = c.iterator();
    { it.ready }
if ( it.hasNext() ) {
        { it.readyForNext }
    MutableInteger x = (MutableInteger) it.next();
        { it.readyForRemove<x> * x.space<1> }
    x.set(42);
        { it.readyForRemove<x> * x.space<1> }
        { it.ready } (by Iterator axiom)
}   { it.ready }
MutableInteger y;
    { it.ready * y.space<1> } (using (Ax Null))
if ( it.hasNext() ) {
        { it.readyForNext * y.space<1> }
    y = (MutableInteger) it.next();
        { it.readyForRemove<y> * y.space<1> }
    it.remove();
        { it.ready * y.space<1> }
}   { it.ready * y.space<1> }
    { c.space<1> * y.space<1> } (by Iterator axiom)
```

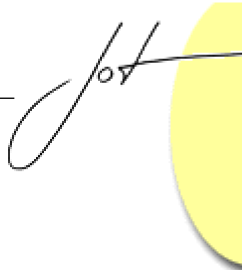
Concurrent read-only iterators:

```
Collection c = new List();
    { c.space<1> }
MutableInteger i = new MutableInteger(0);
    { i.space<1> * c.space<1> }
c.add(i);
    { c.space<1> }
    { c.space<1/2> * c.space<1/2> } (by Object axiom)
Iterator<1/2,c> it1 = c.iterator();
    { it1.ready * c.space<1/2> }
Iterator<1/2,c> it2 = c.iterator();
    { it1.ready * it2.ready }
if ( it1.hasNext() & it2.hasNext() ) {
        { it1.readyForNext * it2.readyForNext }
    MutableInteger x1 = (MutableInteger) it1.next();
    MutableInteger x2 = (MutableInteger) it2.next();
        { it1.readyForRemove<x1> * x1.space<1/2> *
          it2.readyForRemove<x2> * x2.space<1/2> }
    x1.get(); x2.get();
        { it1.readyForRemove<x1> * x1.space<1/2> *
          it2.readyForRemove<x2> * x2.space<1/2> }
        { it1.ready * it2.ready } (by Iterator axiom)
}   { it1.ready * it2.ready }
    { c.space<1/2> * c.space<1/2> } (by Iterator axiom)
    { c.space<1> } (by Object axiom)
```

Iterator over a collection of immutable elements:

```
Collection c = new List();
    { c.space<1> }
Integer zero = new Integer(0);
    { c.space<1> * zero.space<1> }
    { c.space<1> * zero.space<1/2> * zero.space<1/2> } (by Object axiom)
    { c.space<1> * zero.space<1> * zero.space<1> } (by Integer axiom)
c.add(zero);
    { c.space<1> * zero.space<1> }
Iterator<1,c> it = c.iterator();
    { it.ready * zero.space<1> }
Integer x;
    { it.ready * zero.space<1> * x.space<1> } (using (Ax Null))
if ( it.hasNext() ) {
        { it.readyForNext * zero.space<1> * x.space<1> }
    x = (Integer) it.next();
        { it.readyForRemove<x> * zero.space<1> * x.space<1> }
    x.get(); zero.get();
        { it.readyForRemove<x> * zero.space<1> * x.space<1> }
        { it.ready * zero.space<1> * x.space<1> } (by Iterator axiom)
}   { it.ready * zero.space<1> * x.space<1> }
    { c.space<1> * zero.space<1> * x.space<1> } (by Iterator axiom)
```

Figure 3: Instances of Protocol 1



```
class Object {
    //@ pred space<perm p> = true;      // mutable part
    //@ axiom space<p> *-* (space<p/2> * space<p/2>);
    //@ pred ispace = true;             // immutable part
    //@ axiom ispace -* (ispace * ispace);
}
final class MapEntry {
    private int key;   private int val;
    //@ pred ispace = (ex perm p)(key $\stackrel{p}{\longmapsto}$ int);
    //@ pred space<perm p> = val $\stackrel{p}{\longmapsto}$ int
    //@ ens ispace * space<1>;
    public MapEntry(int key, int val) { this.key = key; this.val = val; }
    //@ req ispace;
    public getKey() { return key; }
    //@ <perm p> req space<p>; ens space<p>;
    public getVal() { return val; }
    //@ req space<1>; ens space<1>;
    public setVal(int x) { val = x; }
}
```

Figure 4: Partitioning the `Object` space into mutable and immutable part

Note that `c.add(e)` *only* consumes the access right for `e` if `isdeep==true`.

The iterator protocol associated with a deep collection (as obtained by instantiating the parameter `isdeep` of the `Iterator` type to `true`) is essentially identical to Protocol 1. For shallow collections (obtained by instantiating `isdeep` to `false`), iterators are never allowed to access the mutable part of element states through iterators:

```
interface
Iterator/*@<perm p, boolean isdeep, Collection<isdeep> iteratee>@*/ {
    //@ pred ready;
    //@ pred readyForNext;
    //@ pred readyForRemove<Object element>;
    //@ axiom ready -* iteratee.space<p>;
    //@ axiom readyForRemove<e> * (isdeep -* e.space<p>) -* ready;
    //@ req ready; ens ready & (result -* readyForNext);
    boolean hasNext();
    //@ req readyForNext;
    //@ ens readyForRemove<result> * result.ispace
    //@    * (isdeep -* result.space<p>);
    Object next();
    //@ req readyForRemove<Object> * p==1; ens ready;
    void remove();
}
```

Figure 5 shows a client that creates a shallow collection `c` of map entries, and an iterator over `c`. The collection has type `Collection<false>` and the iterator has type `Iterator<1,false,c>`. The client mutates the value of one of the map entries `e` through a direct reference to `e`, while simultaneously removing `e` from the collection through the

```
Collection<false> c = new List<false>();
    { c.space<1> }
MapEntry e = new MapEntry(0,42);
    { c.space<1> * e.space<1> * e.ispace }
    { c.space<1> * e.space<1> * e.ispace * e.ispace } (by Object axiom for ispace)
c.add(e)
    { c.space<1> * e.space<1> * e.ispace }
Iterator<1,false,c> it = c.iterator();
    { it.ready * e.space<1> * e.ispace }
if ( it.hasNext() ) {
        { it.readyForNext * e.space<1> * e.ispace }
    MapEntry x = (MapEntry) it.next();
        { it.readyForRemove<x> * x.ispace * e.space<1> * e.ispace }
    e.getKey(); e.setVal(21);
    if (x.getKey() == 0) { it.remove(); }
        { it.ready * x.ispace * e.space<1> * e.ispace }
}   { it.ready * e.space<1> * e.ispace }
    { c.space<1> * e.space<1> * e.ispace }
```

Figure 5: An instance of Protocol 2: a shallow iterator

iterator.

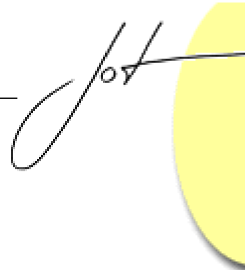
Protocol 3 — Permission-parametrized Iterator States

A slightly more flexible protocol parametrizes the iterator states instead of the iterator type. With this parametrization, we can postulate the following additional iterator axiom:

```
iteratee.space<1/2> * ready<1/2> -* ready<1>
```

This axiom allows converting a read-only iterator to a read-write iterator when other concurrent read-only iterators have terminated. Such a policy is somewhat closer to the policy that Java's library implementations of iterators enforce dynamically.

```
interface Collection {
    //@ <perm p> req space<p>; ens result.ready<p>;
    Iterator/*@<this>@*/ iterator();     }
interface Iterator/*@<Collection iteratee>@*/ {
    //@ pred ready<perm p>;
    //@ pred readyForNext<perm p>;
    //@ pred readyForRemove<perm p, Object element>;
    //@ axiom ready<p> -* iteratee.space<p>;
    //@ axiom readyForRemove<p,e> * e.space<p> -* ready<p>;
    //@ axiom iteratee.space<1/2> * ready<1/2> -* ready<1>;
    //@ <perm p>
    //@ req ready<p>; ens ready<p> & (result -* readyForNext<p>);
    boolean hasNext();
    //@ <perm p>
    //@ req readyForNext<p>; ens readyForRemove<p,result> * result.space<p>;
    Object next();
    //@ req readyForRemove<1,Object>; ens ready<1>;
    void remove();     }
```



```
Collection c = ...
    { c.space<1> }
    { c.space<1/2> * c.space<1/2> } (by Object axiom)
Iterator<c> it1 = c.iterator();
    { it1.ready<1/2> * c.space<1/2> }
Iterator<c> it2 = c.iterator();
    { it1.ready<1/2> * it2.ready<1/2> }
if ( it1.hasNext() & it2.hasNext() ) {
        { it1.readyForNext<1/2> * it2.readyForNext<1/2> }
    MutableInteger x1 = (MutableInteger) it1.next();
    MutableInteger x2 = (MutableInteger) it2.next();
        { it1.readyForRemove<1/2,x1> * x1.space<1/2> *
          it2.readyForRemove<1/2,x2> * x2.space<1/2> }
    x1.get(); x2.get();
        { it1.readyForRemove<1/2,x1> * x1.space<1/2> *
          it2.readyForRemove<1/2,x2> * x2.space<1/2> }
        { it1.ready<1/2> * it2.ready<1/2> } (by Iterator axiom)
}   { it1.ready<1/2> * it2.ready<1/2> }
    { c.space<1/2> * it2.ready<1/2> } (by Iterator axiom)
    { it2.ready<1> } (by the third Iterator axiom)
if ( it2.hasNext() ) {
        { it2.readyForNext<1> }
    MutableInteger x = (MutableInteger) it2.next();
        { it2.readyForRemove<1,x> * x.space<1> }
    x.set(42);
        { it2.readyForRemove<1,x> * x.space<1> }
        { it2.ready<1> } (by Iterator axiom)
}   { it2.ready<1> }
    { c.space<1> } (by Iterator axiom)
```

Figure 6: An Instance of Protocol 3

Like in Protocol 2, we could add a boolean flag as a type parameter in order to support iterators over shallow collections, which we have omitted for simplicity.

Figure 6 shows a client that creates two concurrent read-only iterators `it1` and `it2` over a collection `c`. After `it1` stops iterating, `it2` starts removing and mutating collection elements. Note that, in the third iterator axiom, we have chosen the constant fraction `1/2` instead of stating a similar axiom that uses a permission variable[8]. This *does not mean* that the the protocol only works for two concurrent read-only iterators. The choice of `1/2` merely imposes a certain discipline on the order in which axioms are applied in program proofs. If, for instance, we want to verify a client with three concurrent read-only iterators `it1`, `it2` and `it3`, the third of which becomes read-write after `it1` and `it2` stop iterating, the program proof needs to assign the fraction `1/2` to `it3`. Then, when `it1` and `it2` are in state `ready<1/4>` and `it3` is in state `ready<1/2>`, we can first apply the first iterator axiom to `it1` and `it2` in order to obtain `c.space<1/4> * c.space<1/4>`, then apply the split/merge axiom in `Object` to obtain `c.space<1/2>`, and finally apply the third iterator axiom to obtain `it3.ready<1>`.

[8] In the proof of the iterator implementation, we make use of the fact that we formulate the axiom for $\frac{1}{2}$. See proof of Lemma 6.

```
final class Node {
    /*@ spec_public @*/ Object val;
    /*@ spec_public @*/ Node next;
    //@ spec_public pred space<perm p> =
    //@   valspace<p,this> * nextspace<p,this>;
    //@ <perm p> req _val.space<p> * _next.space<p>; ens space<p>;
    Node(Object _val, Node _next) { val = _val; next = _next; }
    //@ req val<1,_>; ens val<1,_val>;
    public void setVal(Object _val) { val = _val; }
    //@ req next<1,_>; ens next<1,_next>;
    public void setNext(Node _next) { next = _next; }
    //@ <perm p, Object x> req val<p,x>; ens val<p,x> * result==x;
    public Object getVal() { return val; }
    //@ <perm p, Node x> req next<p,x>; ens next<p,x> * result==x;
    public Node getNext() { return next; }
}
```

Figure 7: The `Node` class

4 Iterator Implementations

We provide linked list implementations for Protocols 1 and 3, and sketch the proofs that the implementations satisfy their interfaces. The method implementations are identical in both cases, but the predicate definitions differ. We omit the implementation of Protocol 2, because its proof is not essentially different from Protocol 1, but heavier in notation.

The `Node` class[9] is shown in Figure 7. It makes use of the `spec_public` modifier for fields and predicates, which is syntactic sugar:

- Declaring a field f `spec_public` introduces a predicate f<p,x>, where p is the access permission for this field and x is the value contained in f:

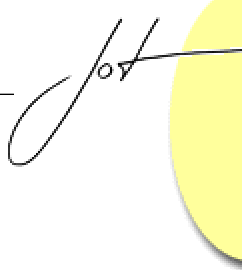
$$\texttt{spec_public}\ T\ f; \quad \triangleq \quad \begin{array}{l} T\ f; \\ \texttt{pred}\ f\texttt{<perm}\ p, T\ x\texttt{>=this.}f \overset{p}{\mapsto} x; \\ \texttt{axiom}\ f\texttt{<}p,x\texttt{>}\ \texttt{*-*}\ \texttt{this.}f \overset{p}{\mapsto} x; \end{array}$$

- Declaring a predicate `spec_public` exports its definition as an axiom. For predicate definitions in class C extending D:

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

In the definition of the `space`-predicate in `Node`, we use helper predicates `valspace`<p,x> and `nextspace`<p,x>. These predicates are defined in Figure 8, together with other helper predicates. The figure associates each helper predicate with a picture and a separation logic formula. The separation logic formula is the official definition, but is really just a textual representation of the depicted heap space.

[9] For Protocol 2, `Node` would need to be parametrized by a boolean flag `isdeep`.



Legend: non-null object *o* — possibly-null object *o* — field *o.f* — *o*.space — object reference *o*

valspace<perm *p*, Node *x*> : (ex Object *o*) (*x*.val<*p*, *o*> * *o*.space<*p*>)

nextspace<perm *p*, Node *x*> : (ex Node *n*) (*x*.next<*p*, *n*> * *n*.space<*p*>)

tail<perm *p*, Node *x*> : *x* != null * *x*.space<*p*>

tail1<perm *p*, Node *x*, Node *y*> : *x* != null * valspace<*p*, *x*> * *x*.next<*p*, *y*> * *y*.space<*p*>

tail2<perm *p*, Node *x*, Node *y*> : *x* != null * valspace<*p*, *x*> * *x*.next<*p*, *y*> * tail<*p*, *y*>

tail3<perm *p*, Node *x*, Node *y*, Node *z*> : *x* != null * valspace<*p*, *x*> * *x*.next<*p*, *y*> * tail1<*p*, *y*, *z*>

tail4<perm *p*, Node *x*, Node *y*, Node *z*, Object *e*> : *x* != null * valspace<*p*, *x*> * *x*.next<*p*, *y*> * *y* != null * *y*.val<*p*, *e*> * *y*.next<*p*, *z*> * z.space<*p*>

Figure 8: Helper predicates

The following abbreviation is also handy. Intuitively, the formula $rest(p,x,C)$ characterizes the heap space difference between x.space<p> (i.e., the heap space associated with object x down to its dynamic class) and x.space@C<p> (i.e., the heap space associated with object x down to class C). In particular, we have (x.space@C<p> * $rest(p,x,C)$) -* x.space<p>, by the linear modus ponens.

$$rest(p,x,C) \triangleq x.\texttt{space@}C\texttt{<}p\texttt{>} \texttt{ -* } x.\texttt{space<}p\texttt{>}$$

Lemma 1

(a) {tail<p,x>}y=x.getNext();{tail1<p,x,y>}
(b) {tail2<p,x,y>}z=y.getNext();{tail3<p,x,y,z>}
(c) {tail3<p,x,y,z>}e=y.getVal();{tail4<p,x,y,z,e> * e.space<p>}
(d) {tail4<1,x,y,z,_>}x.setNext(z);{tail1<1,x,z>}

Proof. By Hoare rules. □

Lemma 2

(a) tail1<p,x,_> -* x.space<p>
(b) tail1<p,x,y> * y != null -* tail2<p,x,y>
(c) tail4<p,x,y,z,e> * e.space<p> -* tail3<p,x,y,z>

Proof. By natural deduction rules. □

The following predicate represents the difference between y.space<p> and x.space<p>:

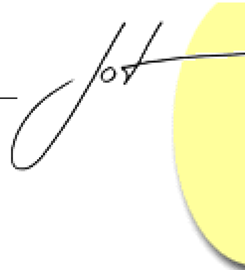
$$\texttt{diff<perm } p\texttt{, Object } y\texttt{, Object } x\texttt{>} \triangleq x.\texttt{space<}p\texttt{>} \texttt{ -* } y.\texttt{space<}p\texttt{>}$$

Lemma 3 x.space<p> * diff<p,y,x> -* y.space<p>

Implementing Interface 1

Figure 9 shows an implementation of the iterator interface for Protocol 1. Iterators have read/write access to their own fields prev, cur and next, hence the access permission 1 for these fields the predicate definitions. The remainder of the predicate definitions are best understood when matched with the pictures of the helper predicates (Figure 8). Note that we have declared the ListIterator class final. Consequently, predicates of the form $v.P$@ListIterator<$\bar{\pi}$> are equivalent to $v.P$<$\bar{\pi}$>. Our proofs make use of this property when establishing abstract predicates in postconditions. If ListIterator were not final, we would have to qualify predicates in postconditions by the class ListIterator (unless the precondition requires the same predicate at method entry).

Figure 10 shows the proof outline for next(). We first translate the method body to a form where intermediate values are assigned to read-only variables, because our Hoare rules are formulated for such a program representation. The proof for next() is straightforward given Lemma 1.



```
class List implements Collection {
    /*@ spec_public @*/ Node header;
    //@ spec_public pred space<perm p> = (ex Node x)(
    //@   header<p,x> * tail<p,x>);
    //@ ens space<1>;
    public List() { header = new Node(null,null); }
    //@ <perm p, Node x> req header<p,x>; ens header<p,x> * result==x;
    Node getHeader() { return header; }    // a helper method
    //@ <perm p> req space<p>; ens result.ready;
    public Iterator/*@<p,this>@*/ iterator() {
        return new ListIterator/*@<p,this>@*/(this);
    }
}

final class ListIterator/*@<perm p, Collection iteratee>@*/
    implements Iterator/*@<p,iteratee>@*/
{
    /*@ spec_public @*/ Node next, cur, prev;
    //@ pred ready =  prev<1,Node> * (ex Node y,z)(
    //@   cur<1,y> * next<1,z> * tail1<p,y,z> * diff<p,iteratee,y>);
    //@ pred readyForNext =  prev<1,Node> * (ex Node y,z)(
    //@   cur<1,y> * next<1,z> * tail2<p,y,z> * diff<p,iteratee,y>);
    //@ pred readyForRemove<Object e> =  (ex Node x,y,z)(
    //@   prev<1,x> * cur<1,y> * next<1,z> *
    //@   tail4<p,x,y,z,e> * diff<p,iteratee,x>);
    //@ req iteratee.space<p> * list==iteratee; ens ready<p>;
    ListIterator(List list) {
        cur = list.getHeader(); next = cur.getNext();
    }
    //@ req ready; ens ready & (result -* readyForNext);
    public boolean hasNext() {
        return next != null;
    }
    //@ req readyForNext; ens readyForRemove<result> * result.space<p>;
    public Object next() {
        prev = cur; cur = next; next = next.getNext();
        return cur.getVal();
    }
    //@ req readyForRemove<Object> * p==1; ens ready;
    public void remove() {
        prev.setNext(next); cur = prev;
    }
}
```

Figure 9: An implementation for Protocol 1

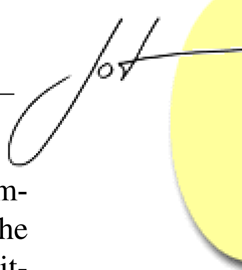
Expanded method body with pre/postcondition:

```
    { readyForNext }
i₁=cur;
prev=i₁;
i₂=next;
cur=i₂;
i₃=next.getNext();
next=i₃;
result=i₂.getVal();
    { readyForRemove<result> * result.space<p> }
```

Proof outline:

```
  { readyForNext }
∴ (by definition of readyForNext@ListIterator)
  { prev<1,Node> * cur<1,y> * next<1,z> * tail2<p,y,z> * diff<p,iteratee,y> }
i₁=cur;
  { prev<1,Node> * cur<1,i₁> * next<1,z> * tail2<p,i₁,z> * diff<p,iteratee,i₁> }
prev=i₁;
  { prev<1,i₁> * cur<1,i₁> * next<1,z> * tail2<p,i₁,z> * diff<p,iteratee,i₁> }
i₂=next;
  { prev<1,i₁> * cur<1,i₁> * next<1,i₂> * tail2<p,i₁,i₂> * diff<p,iteratee,i₁> }
cur=i₂;
  { prev<1,i₁> * cur<1,i₂> * next<1,i₂> * tail2<p,i₁,i₂> * diff<p,iteratee,i₁> }
i₃=next.getNext();   (by Lemma 1(b))
  { prev<1,i₁> * cur<1,i₂> * next<1,i₂> * tail3<p,i₁,i₂,i₃> * diff<p,iteratee,i₁> }
next=i₃;
  { prev<1,i₁> * cur<1,i₂> * next<1,i₃> * tail3<p,i₁,i₂,i₃> * diff<p,iteratee,i₁> }
result=i₂.getVal();   (by Lemma 1(c))
  { prev<1,i₁> * cur<1,i₂> * next<1,i₃> * tail4<p,i₁,i₂,i₃,result> *
    diff<p,iteratee,i₁> * result.space<p> }
∴ (by definition of readyForRemove@ListIterator<result>)
  { result.space<p> * readyForRemove@ListIterator<result> }
∴ (because ListIterator is final)
  { result.space<p> * readyForRemove<p,result> }
```

Figure 10: Proof outline for next()



The proofs for `hasNext()` and `remove()` are similarly straightforward, given Lemmas 1 and 2, and so is the proof of the first iterator axiom. All of these proofs leave the `diff`-predicate untouched. The `diff`-predicate has to be "opened" in the proof of the iterator's constructor (where the predicate is established), and in the proof of the axiom that represents the dashed state transition from `readyForRemove` back to `ready` (where the third argument of `diff` gets modified). Part (a) of the following lemma is what is needed to prove the constructor, and (b) to prove the "`readyForRemove-to-ready`" axiom.

Lemma 4

(a) $c.\texttt{header}<p,h> * h \texttt{ !=null} * rest(p,c,\texttt{List}) \;\texttt{-*}\; \texttt{diff}<p,c,h>$
(b) $(\texttt{tail3}<p,x,y,z> * \texttt{diff}<p,c,x>) \;\texttt{-*}\; (\texttt{tail1}<p,y,z> * \texttt{diff}<p,c,y>)$

Proof. By natural deduction. We provide details for the proof of part (b): By expanding the definitions of `tail3` and `space`, we obtain the following implications:

$$\texttt{tail3}<p,x,y,z> \;\texttt{-*}\; (\texttt{valspace}<p,x> * x.\texttt{next}<p,y> * \texttt{tail1}<p,y,z>) \tag{1}$$
$$(\texttt{valspace}<p,x> * x.\texttt{next}<p,y> * y.\texttt{space}<p>) \;\texttt{-*}\; x.\texttt{space}<p> \tag{2}$$

The following formulas can be verified by natural deduction:

$$A * (B \;\texttt{-*}\; C) * (A \;\texttt{-*}\; A') * (B' \;\texttt{-*}\; B) \;\texttt{-*}\; A' * (B' \;\texttt{-*}\; C) \tag{3}$$
$$A * D * (D * B \;\texttt{-*}\; C) \;\texttt{-*}\; A * (B \;\texttt{-*}\; C) \tag{4}$$

Now suppose that:

$$\texttt{tail3}<p,x,y,z> * \texttt{diff}<p,c,x>$$

Recall that $\texttt{diff}<p,c,x>$ is defined as $x.\texttt{space}<p> \;\texttt{-*}\; c.\texttt{space}<p>$. Using (1), (2) and (3), it follows that:

$$\begin{array}{l}\texttt{valspace}<p,x> * x.\texttt{next}<p,y> * \texttt{tail1}<p,y,z> \\ * (\texttt{valspace}<p,x> * x.\texttt{next}<p,y> * y.\texttt{space}<p> \;\texttt{-*}\; c.\texttt{space}<p>)\end{array}$$

Applying (4), we then obtain:

$$\texttt{tail1}<p,y,z> * (y.\texttt{space}<p> \;\texttt{-*}\; c.\texttt{space}<p>)$$

But this is equivalent to $\texttt{tail1}<p,y,z> * \texttt{diff}<p,c,y>$, by definition of `diff`. □

Implementing Interface 3

Recall that, whereas Protocol 1 permission-parametrizes the iterator interface, Protocol 3 parametrizes the iterator states instead. The slightly modified parametrization does not break any of our proofs for the implementation of Protocol 1. However, we need to refine the predicate definitions in order to be able to prove the the additional iterator axiom:

```
iteratee.space<1/2> * ready<1/2> -* ready<1>                    (5)
```

To this end, we define the following auxiliary combinators:

$$\begin{aligned}
&\mathit{double}(F) \triangleq F * F\\
&\mathit{bump}(p,c,y) \triangleq p\,\texttt{!=}1 * c.\texttt{space<}p\texttt{>} \texttt{ -* } \mathit{double}(\texttt{tail<}p,y\texttt{>} * \texttt{diff<}p,c,y\texttt{>})\\
&\mathit{bump}'(p,c,y,z) \triangleq p\,\texttt{!=}1 * c.\texttt{space<}p\texttt{>} \texttt{ -* } \mathit{double}(\texttt{tail1<}p,y,z\texttt{>} * \texttt{diff<}p,c,y\texttt{>})\\
&\mathit{maybump}(F,p,c,y) \triangleq (F * \texttt{diff<}p,c,y\texttt{>}) \;\texttt{\&}\; \mathit{bump}(p,c,y)
\end{aligned}$$

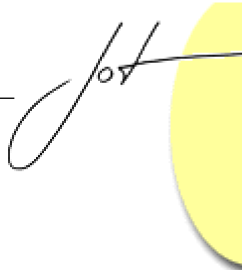
Intuitively, the *maybump*-combinator provides the choice to either keep iterating normally (first factor), or else pay c.space<p> in order to double the permission p associated with the iterator (second factor). We define the iterator predicates as follows:

```
final class ListIterator/*@<Collection iteratee>@*/
    implements Iterator/*@<iteratee>@*/
{
    private Node next, cur, prev;
    //@ pred ready<perm p> =  prev<1,Node> * (ex Node y,z)(
    //@   cur<1,y> * next<1,z> * maybump(tail1<p,y,z>, p, iteratee, y) );
    //@ pred readyForNext<perm p> =  prev<1,Node> * (ex Node y,z)(
    //@   cur<1,y> * next<1,z> * maybump(tail2<p,y,z>, p, iteratee, y) );
    //@ pred readyForRemove<perm p, Object e> =  (ex Node x,y,z)(
    //@   prev<1,x> * cur<1,y> * next<1,z> *
    //@   maybump(tail4<p,x,y,z,e>, p, iteratee, x) );
    ...
}
```

With these refined predicate definitions, class axiom (5) is readily proven. It is a consequence of Lemma 6 below.

Lemma 5 (tail1<p,y,z> & $\mathit{bump}(p,c,y)$) -* $\mathit{bump}'(p,c,y,z)$

Proof. Observe that tail<p,y> implies (ex Node z)(tail1<p,y,z>). The lemma follows from this observation, using (Ax Share). □



Lemma 6 $c.\mathtt{space}\texttt{<}\frac{1}{2}\texttt{>} * maybump(\mathtt{tail1}\texttt{<}\frac{1}{2}, y, z\texttt{>}, \frac{1}{2}, c, y)$
$\texttt{-*}\ maybump(\mathtt{tail1}\texttt{<}1, y, z\texttt{>}, 1, c, y)$

Proof. By substituting the definition of *maybump*, we need to show:

$$c.\mathtt{space}\texttt{<}\tfrac{1}{2}\texttt{>} * ((\mathtt{tail1}\texttt{<}\tfrac{1}{2}, y, z\texttt{>} * \mathtt{diff}\texttt{<}\tfrac{1}{2}, c, y\texttt{>})\ \texttt{\&}\ bump(\tfrac{1}{2}, c, y))$$
$$\texttt{-*}\ ((\mathtt{tail1}\texttt{<}1, y, z\texttt{>} * \mathtt{diff}\texttt{<}1, c, y\texttt{>})\ \texttt{\&}\ bump(1, c, y))$$

By Lemma 5, it then suffices to show:

$$c.\mathtt{space}\texttt{<}\tfrac{1}{2}\texttt{>} * bump'(\tfrac{1}{2}, c, y, z)$$
$$\texttt{-*}\ ((\mathtt{tail1}\texttt{<}1, y, z\texttt{>} * \mathtt{diff}\texttt{<}1, c, y\texttt{>})\ \texttt{\&}\ bump(1, c, y))$$

Observe that $bump(1, c, y)$ holds vacuously, because it is defined as an implication whose antecedent is false. Therefore, it suffices to show:

$$c.\mathtt{space}\texttt{<}\tfrac{1}{2}\texttt{>} * bump'(\tfrac{1}{2}, c, y, z)$$
$$\texttt{-*}\ (\mathtt{tail1}\texttt{<}1, y, z\texttt{>} * \mathtt{diff}\texttt{<}1, c, y\texttt{>})$$

Substituting the definition of $bump'$, it suffices to show:

$$c.\mathtt{space}\texttt{<}\tfrac{1}{2}\texttt{>} * (c.\mathtt{space}\texttt{<}\tfrac{1}{2}\texttt{>}\ \texttt{-*}\ double(\mathtt{tail1}\texttt{<}\tfrac{1}{2}, y, z\texttt{>} * \mathtt{diff}\texttt{<}\tfrac{1}{2}, c, y\texttt{>}))$$
$$\texttt{-*}\ (\mathtt{tail1}\texttt{<}1, y, z\texttt{>} * \mathtt{diff}\texttt{<}1, c, y\texttt{>})$$

By modus ponens, it then suffices to show:

$$double(\mathtt{tail1}\texttt{<}\tfrac{1}{2}, y, z\texttt{>} * \mathtt{diff}\texttt{<}\tfrac{1}{2}, c, y\texttt{>})$$
$$\texttt{-*}\ (\mathtt{tail1}\texttt{<}1, y, z\texttt{>} * \mathtt{diff}\texttt{<}1, c, y\texttt{>})$$

This can easily be shown from the definitions of `tail1` and `diff` by splitting and merging fractions. □

The proofs of `next()`, `hasNext()`, `remove()` and the first iterator axiom are as for Protocol 1, because these proofs only touch the parts of the predicate definitions that coincide for both protocols. In order to prove the constructor and the "`readyForRemove`-to-`ready`" axiom, we have to modify Lemma 4 appropriately:

Lemma 7

(a) $c.\mathtt{header}\texttt{<}p, h\texttt{>} * \mathtt{tail}\texttt{<}p, h\texttt{>} * rest(p, c, \mathtt{List})\ \texttt{-*}\ bump(p, c, h)$
(b) $maybump(\mathtt{tail3}\texttt{<}p, x, y, z\texttt{>}, p, c, x)\ \texttt{-*}\ maybump(\mathtt{tail1}\texttt{<}p, y, z\texttt{>}, p, c, y)$

Proof. By natural deduction. We sketch details for the proof of part (b): By Lemma 4(b), it suffices to show the following:

$$(\ (\texttt{tail3<}p,x,y,z\texttt{>} * \texttt{diff<}p,c,x\texttt{>})\ \&\ \mathit{bump}(p,c,x)\ )\ \texttt{-*}\ \mathit{bump}(p,c,y)$$

So suppose:

$$(\texttt{tail3<}p,x,y,z\texttt{>} * \texttt{diff<}p,c,x\texttt{>})\ \&\ \mathit{bump}(p,c,x)$$

By expanding *bump* and then `tail`, and then using axiom (Ax Share), we obtain:

$$p\ \texttt{!=}\ 1 * c.\texttt{space<}p\texttt{>}\ \texttt{-*}\ \mathit{double}(\texttt{tail1<}p,x,y\texttt{>} * y\ \texttt{!=null} * \texttt{diff<}p,c,x\texttt{>})$$

By similar reasoning as in the proof of Lemma 4(b), we then obtain:

$$p\ \texttt{!=}\ 1 * c.\texttt{space<}p\texttt{>}\ \texttt{-*}\ \mathit{double}(y.\texttt{space<}p\texttt{>} * y\ \texttt{!=null} * \texttt{diff<}p,c,y\texttt{>})$$

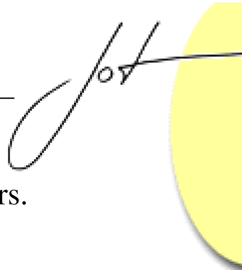
But this is the same as $\mathit{bump}(p,c,y)$, by definitions of `tail` and *bump*. □

5 Related Work

Recently, iterators have served as a challenging case study for several verification systems, namely, separation logic [Par05], higher-order separation logic [Kri06], a linear typestate system [Bie06, BA07], and a linear type-and-effect system [BRZ07].

Parkinson [Par05] uses iterators as an example. He supports simultaneous read-only iterators through counting permissions, rather than fractional permissions. He considers iterators over shallow collections, but unlike us does not consider deep collections as well. His iterator interface does not have a `remove()` method, which is particularly interesting for deep collections, because it is important that the collection passes the access permission for removed elements to the remover. Parkinson's proof of his iterator implementation is different from ours: it uses a lemma that is proven inductively, whereas our proof is based on the natural deduction rules for linear logic without an induction rule.

Krishnaswami [Kri06] presents a protocol for iterators over linked lists using *higher-order* separation logic. His collections are shallow and his iterators are read-only. His protocol allows multiple iterators over a collection and enforces that all active iterators are abandoned, once a new element is added to the collection. Technically, he achieves this by parametrizing a (higher-order) predicate by a (first-order) predicate that represents the state of a collection. Iterators that are created when the underlying collection is in a certain state P may only be used as long as the collection is still in state P. The `add()`



method does not preserve the collection state and hence invalidates all existing iterators. This is a very elegant solution that makes use of the power of higher-order predicates.

Bierhoff and Aldrich [BA07] present a linear typestate system based on a fragment of linear logic, which includes multiplicative conjunction, additive conjunction and additive disjunction. It uses $\multimap$ as a separator between pre- and postconditions (but not as a logical connective that represents linear implication). Bierhoff and Aldrich use iterators as a case study for their system [BA07, Bie06]. They support concurrent read-only iterators through fractional permissions. Their protocols do not not support iterators over deep collections with *mutable* collection elements, although [Bie06] supports *read-only* access to collection elements that get returned by `next()`. Our protocol cannot be represented in their system because their specification language lacks linear implication (needed to represent the dashed `readyForRemove`-to-`ready` transition). Of course, they could add linear implication to their language. They associate our second dashed transition (the one that terminates an iteration) with the iterator's `finalize()` method, and assume that a checker would employ program analysis techniques to apply the `finalize()`-contract without explicitly calling `finalize()`. In practice, this has the same effect as our first iterator axiom. Neither of [BA07, Bie06] presents an iterator implementation, or a mapping of iterator state predicates to concrete definitions. To verify iterator implementations, related verification systems employ recursive predicates and either induction [Par05] or introduction and elimination rules for linear implication (this article) or both [Kri06]. Recursive predicates, induction and linear implication are not supported by [BA07, Bie06], and it is thus possible that [BA07, Bie06] is not expressive enough to verify iterator implementations. Of course, it is likely that Bierhoff and Aldrich have deliberately avoided such features (especially induction) because their system is designed as a lightweight typestate system, rather than a full-blown program logic. As a minor remark, we point out that a protocol like our Protocol 3 (on page 66) is not expressible in Bierhoff and Aldrich's system, as discussed in [Bie06]. Protocol 3 allows read-only iterators to turn into read-write iterators when all other concurrent iterators over the same collection have terminated. This is also allowed by the iterator specifications in [Par05] and [Kri06].

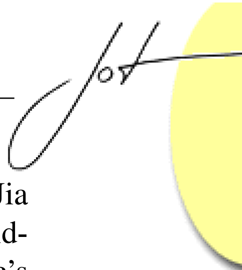
Boyland, Retert and Zhao [BRZ07] informally explain how to apply their linear type and effect system (an extension of [BR05] with fractional permissions) to specify and verify iterator protocols. Their system facilitates concurrent read-only iterators through fractional permissions. The paper does not address iterators over deep collections. In contrast to [BRZ07]'s `Iterator` interface, our `Iterator` interface is parametrized by the underlying collection. As a result, in our system client methods and classes sometimes need an auxiliary parameter (in angle brackets). Like us, [BRZ07] use linear implication to represent the state transition that finalizes an iterator. They represent this linear implication as an effect on the `iterator()` method, whereas we choose to represent it as a class axiom. Their linear implication operator (called "scepter") has a different semantics than separation logic's magic wand, which we use[10].

[10] A formal semantics for their system is presented in [Boy07], where Boyland explains that he wants a more precise semantics of linear implication. For instance, the following formula is a semantic tautology both for the magic wand and Boyland's scepter: $x.f \mapsto 3$ `-*` $y.f \mapsto 5$ `-*` $(x.f \mapsto 3 * y.f \mapsto 5)$. However, the following formula is a semantic tautology for the magic wand but not for Boyland's scepter: $y.f \mapsto$

Compared to the related work discussed above, a technical contribution of this article is support for *iterators over deep collections*. None of the protocols above allows read/write access to collection elements that get returned by `next()`. Our protocols can grant read/write access to such collection elements, and keep access to these elements under control by requiring that access permissions get abandoned before the next collection elements can be retrieved. We do not claim that the other logics cannot, in principle, express protocols for iterators over deep collections, but rather that other case studies with iterators did not consider deep collections. In principle, iterator protocols over deep collections are also expressible in separation logic with fractional permissions ([BOCP05]), as our verification system is essentially a subsystem of separation logic with fractional permissions. Furthermore, we believe that iterator protocols over deep collections are expressible in *higher-order* separation logic (without fractional permissions), using Krishnaswami's technique that employs higher-order predicates for read-sharing [Kri06]. As discussed above, while Bierhoff and Aldrich's linear typestate system [BA07] is suitable for verifying iterator clients, it may not be expressive enough to verify iterator implementations (both over deep and shallow collections), as it lacks recursive predicates and either induction or linear implication. However, we believe that, if recursive predicates and linear implication were added to that system and if our class axioms for `Iterator` were replaced by richer postconditions as discussed on page 63, then Protocol 1 (page 61) for deep collections could be expressed and iterator clients and implementations could be verified in Bierhoff and Aldrich's system.

Our main motivation for defining logical consequence proof-theoretically was that this seemed more amenable to automation than a model-theoretical definition. It is therefore worthwhile discussing and comparing to work on automatic assertion checking for separation logic and related formalisms: One such line of work is rooted in the Smallfoot tool [BCO05b, BCO05a], which has been the inspiration for various other research tools [NDQC07, CDNQ08, DP08, JP08]. Among the latter, [CDNQ08] and [DP08] are particularly noteworthy here, as they specifically treat object-oriented languages. Smallfoot restricts logical assertions to a decidable subset of separation logic. It supports a set of built-in recursive predicates for lists and trees. In order to prove interesting programs, several inductively proven lemmas are hard-wired into Smallfoot's checking algorithm, and are verified by hand in the meta-theory. An attractive aspect of our system is that we can prove interesting list-manipulating programs without induction or relying on inductively proven meta-lemmas. Being able to avoid induction seems particularly interesting for automation. In our iterator implementation, the key for avoiding induction is the use of linear implication. While Smallfoot and its above-mentioned successors do not support linear implication, Jia and Walker's Intuitionistic Linear Logic with Constraints (ILC) does [JW06]. ILC is a verification system based on a linear logic proof theory that is sound with respect to the separation logic model and, in that respect, is quite similar to our system. ILC includes a verification condition generator that yields both classical logic verification conditions (discharged by an SMT solver) and linear logic verification

$42 \mathrel{-\!*} y.f \mapsto 5 \mathrel{-\!*} (x.f \mapsto 3 * y.f \mapsto 5)$. We believe that the latter formula is not provable in our proof theory either. Technically, Boyland's semantics for the scepter combines ideas from proof theory and model theory in a single definition.



conditions. The latter are currently solved by an interactive linear logic prover, but Jia and Walker report that research on automation is in progress. They also present a decidable sublanguage ILC$^-$, achieving decidability by syntactically restricting linear logic's !-modality[11]. In particular, ILC$^-$disallows formulas of the shape $!(F \text{ -* } G)$. This means that our `Iterator` axioms do not fall into ILC$^-$, as axioms in linear logic are represented by !-ed formulas (because they can be applied arbitrarily often). As discussed on page 63, we could avoid axioms in the `Iterator` interface, but many of our Java axioms in Appendix A do not fall into ILC$^-$ either. Thus, the automation of (fragments of) our verification system remains interesting future work.

6 Conclusion

We have discussed several `Iterator` usage protocols that prevent concurrent modifications of the underlying collection, and have formalized them in a variant of separation logic. From the point of view of iterator clients these protocols are quite similar to recent protocols expressed in linear typestate systems [Bie06, BA07], but in addition support disciplined use of iterators over deep collections, by employing linear implications to represent state transitions that are not associated with method calls.

Separation logic provides a firm basis for verifying iterator implementations in addition to iterator clients. Standard soundness results for separation logic imply that verified programs satisfy certain global safety properties, notably, that verified multithreaded programs are datarace-free. In particular, concurrent iterations over the same collection cannot result in dataraces.

We note that verifying adherence to iterator usage protocols seems considerably easier than verifying iterator implementations (as already remarked by [Kri06]). This is not surprising, because implementing linked data structures is error prone and certainly much harder than using iterators in a disciplined way. However, separation logic proofs for linked data structures are very concrete and closely related to the kinds of pictures that we all draw when we write pointer programs.

The automation of (fragments of) our verification system remains interesting future work. We will see how far this can be pushed.

Acknowledgments. We thank the reviewers of this JOT article for their very careful reviews that lead to many improvements. Furthermore, we thank Marieke Huisman and Erik Poll for interesting discussions about this work, and the reviewers of the preceding IWACO article for useful comments.

A Verification Rules

Natural Deduction Rules, $\Gamma;\bar{F} \vdash G$**:**

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

(Ax)
$$\frac{\Gamma \vdash F \quad \Gamma \vdash F : \diamond}{\Gamma; \vdash F}$$

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

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

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

[11] The !-modality promotes linear formulas to copyable formulas.

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

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

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

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

(Fa Intro)
$$\frac{x \notin \bar{F} \quad \Gamma,x:T;\bar{F} \vdash G}{\Gamma;\bar{F} \vdash (\texttt{fa}\, T\, x)(G)}$$

(Fa Elim)
$$\frac{\Gamma;\bar{F} \vdash (\texttt{fa}\, T\, x)(G) \quad \Gamma \vdash \pi : T}{\Gamma;\bar{F} \vdash G[\pi/x]}$$

(Ex Intro)
$$\frac{\Gamma,x:T \vdash G:\diamond \quad \Gamma \vdash \pi : T \quad \Gamma;\bar{F} \vdash G[\pi/x]}{\Gamma;\bar{F} \vdash (\texttt{ex}\, T\, x)(G)}$$

(Ex Elim) $x \notin \bar{F}, H$
$$\frac{\Gamma;\bar{E} \vdash (\texttt{ex}\, T\, x)(G) \quad \Gamma,x:T;\bar{F},G \vdash H}{\Gamma;\bar{E},\bar{F} \vdash H}$$

Java Axioms, $\Gamma \vdash F$:

(Ax True)
$$\frac{}{\Gamma \vdash \texttt{true}}$$

(Ax False)
$$\frac{}{\Gamma \vdash \texttt{false} \mathrel{-\!\!*} F}$$

(Ax Pure)
$$\frac{}{\Gamma \vdash (e \,\&\, F) \mathrel{-\!\!*} (e * F)}$$

(Ax Subst)
$$\frac{\Gamma \vdash e,e' : T \quad \Gamma,x:T \vdash F:\diamond}{\Gamma \vdash (F[e/x] * e == e') \mathrel{-\!\!*} F[e'/x]}$$

(Ax Bool)
$$\frac{\Gamma \models \,!e_1 \mid\, !e_2 \mid e'}{\Gamma \vdash (e_1 * e_2) \mathrel{-\!\!*} e'}$$

(Ax Split/Merge)
$$\frac{}{\Gamma \vdash v.f \overset{\pi}{\mapsto} e \mathrel{*\!\!-\!\!*} (v.f \overset{\pi/2}{\longmapsto} e * v.f \overset{\pi/2}{\longmapsto} e)}$$

(Ax Cl) $\Gamma \vdash \pi : t\texttt{<}\bar{\pi}'\texttt{>}$
$$\frac{\mathsf{axiom}(t\texttt{<}\bar{\pi}'\texttt{>}) = F}{\Gamma \vdash F[\pi/\texttt{this}]}$$

(Ax Open/Close) $\Gamma \vdash \texttt{this} : C\texttt{<}\bar{\pi}'\texttt{>}$
$$\frac{\mathsf{pbody}(P\texttt{<}\bar{\pi}\texttt{>}, C\texttt{<}\bar{\pi}'\texttt{>}) = F \quad C\texttt{<}\bar{\pi}'\texttt{>}\ \texttt{extends}\ D\texttt{<}\bar{\pi}''\texttt{>}}{\Gamma \vdash \texttt{this}.P\texttt{@}C\texttt{<}\bar{\pi}\texttt{>} \mathrel{*\!\!-\!\!*} (F * \texttt{this}.P\texttt{@}D\texttt{<}\bar{\pi}\texttt{>})}$$

(Ax Final)
$$\frac{\Gamma \vdash v : C\texttt{<}\bar{\pi}\texttt{>} \quad C \text{ is } \texttt{final}}{\Gamma \vdash C\ \texttt{isclassof}\ v}$$

(Ax Null)
$$\frac{}{\Gamma \vdash \texttt{null}.\kappa\texttt{<}\bar{\pi}\texttt{>}}$$

(Ax Sub Cl)
$$\frac{C \preceq D}{\Gamma \vdash v.P\texttt{@}D\texttt{<}\bar{\pi}\texttt{>}\ \texttt{ispartof}\ v.P\texttt{@}C\texttt{<}\bar{\pi}\texttt{>}}$$

(Ax Sub Dyn)
$$\frac{}{\Gamma \vdash v.P\texttt{@}C\texttt{<}\bar{\pi}\texttt{>}\ \texttt{ispartof}\ v.P\texttt{<}\bar{\pi}\texttt{>}}$$

(Ax Dyn)
$$\frac{}{\Gamma \vdash (v.P\texttt{@}C\texttt{<}\bar{\pi}\texttt{>} * C\ \texttt{isclassof}\ v) \mathrel{-\!\!*} v.P\texttt{<}\bar{\pi}\texttt{>}}$$

(Ax Share)
$$\frac{\text{the hole in } F[\,] \text{ is not to the left of a } \mathrel{-\!\!*}}{\Gamma \vdash (\, v.f \overset{\pi}{\mapsto} e \ \&\ F[(\texttt{ex}\, T\, x)(v.f \overset{\pi'}{\mapsto} x * G)]\,) \mathrel{-\!\!*} F[v.f \overset{\pi'}{\mapsto} e * G[e/x]]}$$

where $\mathsf{axiom}(t\texttt{<}\bar{\pi}\texttt{>}) \triangleq$ $*$-conjunction of all axioms in $t\texttt{<}\bar{\pi}\texttt{>}$ and its supertypes

$\mathsf{pbody}(P\texttt{<}\bar{\pi}\texttt{>}, C\texttt{<}\bar{\pi}'\texttt{>}) \triangleq F$, if F is $P\texttt{<}\bar{\pi}\texttt{>}$'s definition in $C\texttt{<}\bar{\pi}'\texttt{>}$ $\quad$ $\mathsf{pbody}(P\texttt{<}\bar{\pi}\texttt{>}, C\texttt{<}\bar{\pi}'\texttt{>}) \triangleq \texttt{true}$, otherwise

$F[\,]$ is a formula with exactly one "hole": $\quad F[\,] ::= [\,] \mid e \mid v.f \overset{\pi}{\mapsto} e \mid v.\kappa\texttt{<}\bar{\pi}\texttt{>} \mid F[\,]\ lop\ F \mid F\ lop\ F[\,] \mid (qt\ T\ x)(F[\,])$

$F[G]$ is the formula obtained by filling $F[\,]$'s hole with formula G (capturing free variables of G)

We assume that, prior to verification, commands have been transformed to a form, where all intermediate values are assigned to read-only variables (ranged over by x):

$$\begin{array}{rcl} c \in \mathsf{Cmd} & ::= & hc;c \mid v \\ hc \in \mathsf{HdCmd} & ::= & T\,\ell \mid x\texttt{=}\ell \mid \ell\texttt{=}v \mid x\texttt{=}op(\bar{v}) \mid x\texttt{=}v.f \mid v.f\texttt{=}v \mid x\texttt{=new}\,C\texttt{<}\bar{\pi}\texttt{>} \mid x\texttt{=}v.m(\bar{v}) \mid \texttt{if}\,(v)\{c\}\texttt{else}\{c'\} \end{array}$$

Hoare Triples, $\Gamma \vdash \{F\}c:T\{G\}$ and $\Gamma \vdash \{F\}hc\{G\} \dashv \Gamma'$:

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

(Return)
$$\frac{\Gamma \vdash v:T \quad \Gamma,\texttt{result}:T \vdash G:\diamond}{\Gamma \vdash \{G[v/\texttt{result}]\}v:T\{G\}}$$

(Frame) $\mathsf{RdWrVar}(H) \cap \mathsf{Modifies}(hc) = \emptyset$
$$\frac{\Gamma \vdash H:\diamond \quad \Gamma \vdash \{F\}hc\{G\} \dashv \Gamma'}{\Gamma \vdash \{F * H\}hc\{G * H\} \dashv \Gamma'}$$

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

(Aux Var)
$$\frac{\Gamma,x:T \vdash \{F\}hc\{G\} \dashv \Gamma',x:T}{\Gamma \vdash \{(\texttt{ex}\, T\, x)(F)\}hc\{(\texttt{ex}\, T\, x)(G)\} \dashv \Gamma'}$$

(Var Dcl)
$$\frac{\Gamma \vdash T:\diamond}{\Gamma \vdash \{\texttt{true}\}T\,\ell\{\ell == \mathsf{default}(T)\} \dashv \Gamma,\ell:T}$$

(Get Var)
$$\frac{\Gamma \vdash \ell:T}{\Gamma \vdash \{\texttt{true}\}x\texttt{=}\ell\{x == \ell\} \dashv \Gamma,x:T}$$

(Set Var)
$$\frac{\Gamma \vdash v:\Gamma(\ell)}{\Gamma \vdash \{\texttt{true}\}\ell\texttt{=}v\{\ell == v\} \dashv \Gamma}$$

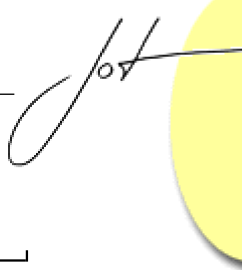
(Op)
$$\frac{\Gamma \vdash op(\bar{v}):T}{\Gamma \vdash \{\texttt{true}\}x\texttt{=}op(\bar{v})\{x == op(\bar{v})\} \dashv \Gamma,x:T}$$

(Get)
$$\frac{\Gamma;F \vdash v.f \overset{\pi}{\mapsto} w \quad \Gamma \vdash v:V \quad W\,f \in \mathsf{fld}(V)}{\Gamma \vdash \{F\}x\texttt{=}v.f\{F * x == w\} \dashv \Gamma,x:W[v/\texttt{this}]}$$

(Set)
$$\frac{\Gamma \vdash v:V \quad W\,f \in \mathsf{fld}(V) \quad \Gamma \vdash w:W[w/\texttt{this}]}{\Gamma \vdash \{v.f \overset{1}{\mapsto} W\}v.f\texttt{=}w\{v.f \overset{1}{\mapsto} w\} \dashv \Gamma}$$

(New) $C\texttt{<}\bar{T}\,\bar{y}\texttt{>}$ is declared $\quad \Gamma \vdash \bar{\pi} : \bar{T}[\bar{\pi}/\bar{y}]$
$$\frac{\texttt{init} = \circledast_{T\, f \in \mathsf{fld}(C\texttt{<}\bar{\pi}\texttt{>})} x.f \overset{1}{\mapsto} \mathsf{default}(T)}{\Gamma \vdash \{\texttt{true}\}x\texttt{=new}\,C\texttt{<}\bar{\pi}\texttt{>}\{\texttt{init} * C\ \texttt{isclassof}\ x\} \dashv \Gamma,x:C\texttt{<}\bar{\pi}\texttt{>}}$$

(If)
$$\frac{\Gamma \vdash v:\texttt{boolean} \quad \Gamma \vdash \{F * v\}c:\texttt{void}\{G\} \quad \Gamma \vdash \{F * !v\}c':\texttt{void}\{G\}}{\Gamma \vdash \{F\}\texttt{if}\,(v)\{c\}\texttt{else}\{c'\}\{G\} \dashv \Gamma}$$



(Call)

$$\frac{\Gamma \vdash v : V \quad \mathsf{mtype}(m,V) = \texttt{<}\bar{T}\,\bar{z}\texttt{>}\texttt{req}\,F; \texttt{ens}\,G; U\,m(\bar{W}\,\bar{y}) \quad \Gamma \vdash \bar{\pi} : \bar{T}[\sigma] \quad \Gamma \vdash \bar{w} : \bar{W}[\sigma] \quad \sigma = (v/\texttt{this}, \bar{\pi}/\bar{z}, \bar{w}/\bar{y})}{\Gamma \vdash \{v\,\texttt{!=null} * F[\sigma]\}x\texttt{=}v.m(\bar{w})\{G[\sigma, x/\texttt{result}]\} \dashv \Gamma, x : U[\sigma]}$$

REFERENCES

[BA07] K. Bierhoff and J. Aldrich. Modular typestate verification of aliased objects. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 301–320, 2007.

[BCO05a] J. Berdine, C. Calcagno, and P. W. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In F. S. de Boer, M. M. Bonsangue, S. Graf, and W.-P. de Roever, editors, *Formal Methods for Components and Objects*, volume 4111 of *Lecture Notes in Computer Science*, pages 115–137. Springer-Verlag, 2005.

[BCO05b] J. Berdine, C. Calcagno, and P. W. O'Hearn. Symbolic execution with separation logic. In K. Yi, editor, *Asian Programming Languages and Systems Symposium*, volume 3780 of *Lecture Notes in Computer Science*, pages 52–68. Springer-Verlag, 2005.

[BDF+04] M. Barnett, R. DeLine, M. Fähndrich, K. R. M. Leino, and W. Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3(6):27–56, 2004.

[Bie06] K. Bierhoff. Iterator specification with typestates. In *Specification and Verification of Component-Based Systems*, pages 79–82, 2006.

[BOCP05] R. Bornat, P. W. O'Hearn, C. Calcagno, and M. Parkinson. Permission accounting in separation logic. In J. Palsberg and M. Abadi, editors, *Principles of Programming Languages*, pages 259–270. ACM Press, 2005.

[Boy03] J. Boyland. Checking interference with fractional permissions. In R. Cousot, editor, *Static Analysis Symposium*, volume 2694 of *Lecture Notes in Computer Science*, pages 55–72. Springer-Verlag, 2003.

[Boy07] J. Boyland. Semantics of fractional permissions with nesting. Technical report, University of Wisconsin at Milwaukee, December 2007.

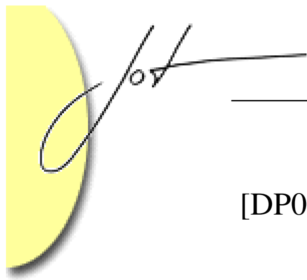
[BR05] J. Boyland and W. Retert. Connecting effects and uniqueness with adoption. In *Principles of Programming Languages*, pages 283–295, 2005.

[BRZ07] J. Boyland, W. Retert, and Y. Zhao. Iterators can be independent "from" their collections. International Workshop on Aliasing, Confinement and Ownership in object-oriented programming, 2007.

[CDNQ08] W. Chin, C. David, H. Nguyen, and S. Qin. Enhancing modular OO verification with separation logic. In G. C. Necula and P. Wadler, editors, *Principles of Programming Languages*, pages 87–99. ACM Press, 2008.

[DF01] R. DeLine and M. Fähndrich. Enforcing high-level protocols in low-level software. In *Programming Languages Design and Implementation*, pages 59–69, 2001.

[DF04] R. DeLine and M. Fähndrich. Typestates for objects. In *European Conference on Object-Oriented Programming*, pages 465–490, 2004.



[DP08] D. DiStefano and M. Parkinson. jStar: Towards practical verification for Java. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, volume 43, pages 213–226. ACM Press, 2008.

[Gir87] J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[Gir95] J.-Y. Girard. Linear logic: Its syntax and semantics. In J.-Y. Girard, Y. Lafont, and L. Regnier, editors, *Advances in Linear Logic (Proc. of the Workshop on Linear Logic, Cornell University, June 1993)*, number 222. Cambridge University Press, 1995.

[HH08a] C. Haack and C. Hurlin. Separation logic contracts for a Java-like language with fork/join. In J. Meseguer and G. Rosu, editors, *Algebraic Methodology and Software Technology*, volume 5140 of *Lecture Notes in Computer Science*, pages 199–215. Springer-Verlag, July 2008.

[HH08b] C. Haack and C. Hurlin. Separation logic contracts for a Java-like language with fork/join. Technical Report 6430, INRIA, January 2008.

[HHH08] C. Haack, M. Huisman, and C. Hurlin. Reasoning about Java's reentrant locks. In G. Ramalingam, editor, *Asian Programming Languages and Systems Symposium*, volume 5356 of *Lecture Notes in Computer Science*, pages 171–187. Springer-Verlag, December 2008.

[IO01] S. Ishtiaq and P. W. O'Hearn. BI as an assertion language for mutable data structures. In *Principles of Programming Languages*, pages 14–26, 2001.

[JP08] B. Jacobs and F. Piessens. The VeriFast program verifier. Technical Report CW-520, Katholieke Universiteit Leuven, 2008.

[JW06] L. Jia and D. Walker. ILC: A foundation for automated reasoning about pointer programs. In *European Symposium on Programming*, pages 131–145, 2006.

[Kri06] G. Krishnaswami. Reasoning about iterators with separation logic. In *Specification and Verification of Component-Based Systems*, pages 83–86, 2006.

[NDQC07] H. H. Nguyen, C. David, S. Qin, and W. Chin. Automated verification of shape and size properties via separation logic. In *Verification, Model Checking and Abstract Interpretation*, pages 251–266, 2007.

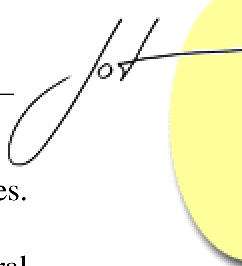
[OP99] P. W. O'Hearn and D. J. Pym. The logic of bunched implications. *Bulletin of Symbolic Logic*, 5(2):215–244, 1999.

[OYR04] P. W. O'Hearn, H. Yang, and J. C. Reynolds. Separation and information hiding. In *Principles of Programming Languages*, pages 268–280. ACM Press, January 2004.

[Par05] M. Parkinson. Local reasoning for Java. Technical Report UCAM-CL-TR-654, University of Cambridge, 2005.

[PB05] M. Parkinson and G. Bierman. Separation logic and abstraction. In J. Palsberg and M. Abadi, editors, *Principles of Programming Languages*, pages 247–258. ACM Press, 2005.

[PBB+04] M. Pavlova, G. Barthe, L. Burdy, M. Huisman, and J.-L. Lanet. Enforcing high-level security properties for applets. In P. Paradinas and J.-J. Quisquater, editors, *Smart Card Research and Advanced Application*. Kluwer Academic Publishing, 2004.



[Rey02] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Logic in Computer Science*. IEEE Press, July 2002.

[TH02] K. Trentelman and M. Huisman. Extending JML specifications with temporal logic. In *Algebraic Methodology and Software Technology*, volume 2422 of *Lecture Notes in Computer Science*, pages 334–348. Springer-Verlag, 2002.

[Wad93] P. Wadler. A taste of linear logic. In *Mathematical Foundations of Computer Science*, pages 185–210, 1993.

About the authors

Christian Haack (chaack@cs.ru.nl) was a researcher in the Digital Security Group at Radboud University Nijmegen in The Netherlands, while doing this research. He now works for aicas realtime — a company that develops Realtime Java technology — in Karlsruhe, Germany. He remains in close touch with Radboud University and has a web page at http://www.cs.ru.nl/˜chaack.

Clément Hurlin (clement.hurlin@sophia.inria.fr) is a PhD student in the Everest team at Inria Sophia-Antipolis in France. He is currently visiting the Formal Methods & Tools group at University of Twente in The Netherlands. He has a web page at http://www-sop.inria.fr/everest/Clement.Hurlin.