Resource Usage Protocols for Iterators

Christian Haack [*]
Radboud University Nijmegen, The Netherlands
chaack@cs.ru.nl

Clément Hurlin [†]
INRIA Sophia Antipolis - Méditerranée, France
clement.hurlin@sophia.inria.fr

Abstract

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 OO languages formally specify type signatures, they usually do not formalize 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 (Pavlova et al. 2004) and temporal logic extensions (Trentelman and Huisman 2002). 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 (DeLine and Fähndrich 2001; DeLine and Fähndrich 2004) and ideas from linear logic (Bierhoff and Aldrich 2007).

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♯. Usage protocols for iterators are meant to prevent so-called *concurrent modifications* of the underlying collection: while an iterator over a collection is in progress, the collection should not get modified by other actions that interleave with the iteration[1]. Concurrent modifications need to be prevented,

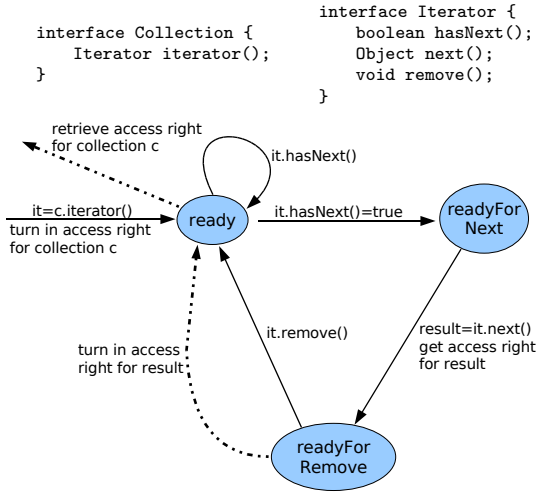

Figure 1. Basic iterator protocol

because they may temporarily break invariants of the collection, resulting in iterators seeing collections in inconsistent states. Concurrent modifications can be caused both by modifications of the collection itself (i.e., adding or removing collection elements), or in some cases by modification of the collection elements (i.e., resetting fields of collection elements). Consider for instance a collection of mutable point objects, subject to the invariant that the sum of all x-coordinates is greater than the sum of all y-coordinates. This invariant could get temporarily broken by removing a point whose x-coordinate is larger than its y-coordinate. It could also get broken by resetting a previously positive x-coordinate to zero.

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()`.

[*] Supported by IST-FET-2005-015905 Mobius project.

[†] Supported by IST-FET-2005-015905 Mobius project and ANR-06-SETIN-010 ParSec project.

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

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

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 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 (Haack and Hurlin 2008b), which is based on intuitionistic separation logic (Ishtiaq and O'Hearn 2001; Reynolds 2002; Parkinson and Bierman 2005). Compared to standard presentations of separation logic, a peculiarity of (Haack and Hurlin 2008b) is that we define logical consequence proof-theoretically, using a natural deduction calculus that is common to the (affine) logic of bunched implication (BI) (O'Hearn and Pym 1999) and (affine) linear logic (Girard 1987). 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 (Wadler 1993), 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 (Girard 1995) 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 (and we tend to think in terms of permissions rather than in terms of heaplets).

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. 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 be used to represent non-deterministic state transitions, as exhibited in the postcondition of `hasNext()`. The *resource implication* F `-*` G grants the right to consume F yielding G. Boolean expressions e are treated as *copyable* resources, i.e., they satisfy e `-*` `(`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 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 (Haack and Hurlin 2008b), 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 (Boyland 2003), which may occur in contracts and as type parameters, but not in executable code.

Values and Specification Values:

$n \in \mathsf{Int}$ $\quad b \in \mathsf{Bool}$ $\quad x \in \mathsf{Var}$

$v \in \mathsf{Val} ::= b \mid n \mid x$ $\quad \pi \in \mathsf{SpecVal} ::= v \mid 1 \mid \texttt{split}(\pi)$

Derived form: $\frac{\pi}{2^n} \triangleq \texttt{split}^n(\pi)$

Fractional permissions are binary fractions in the interval $(0,1]$. Fractional permissions have type `perm`. Predicates and types may be parametrized by fractional permissions. As usual (Bornat et al. 2005), fractional permissions are arguments of the points-to predicate, in order to govern access rights : $v.f \overset{\pi}{\mapsto} e$ 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,

[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` $\models F$ `-*` G. In the implementation of the iterator interface, we use a true resource implication.

[4] We mean *persistent* immutability, in contrast to (Bierhoff and Aldrich 2007) which uses the term "immutable" for temporarily immutable state.

while permitting concurrent reads. The key for flexibly enforcing this global invariant is the *split-merge law*:

$$v.f \overset{\pi}{\mapsto} e \;\texttt{*-*}\; (v.f \overset{\pi/2}{\longmapsto} e * v.f \overset{\pi/2}{\longmapsto} e)$$

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.

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

The *qualified predicate* $v.P\texttt{@}C\texttt{<}\bar{\pi}\texttt{>}$ represents the conjunction of all predicate extensions for P in C's superclasses, up to and including C. The *unqualified predicate* $v.P\texttt{<}\bar{\pi}\texttt{>}$ is equivalent to $v.P\texttt{@}C\texttt{<}\bar{\pi}\texttt{>}$, where C is v's dynamic class. Our structured way of extending predicates facilitates modular verification (preventing re-verification of inherited methods), and is inspired by the so-called "stack of class frames" (DeLine and Fähndrich 2004; Barnett et al. 2004).

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

Specification Formulas:

$$\begin{array}{rrcl}
op \in \mathsf{Op} \supseteq & \{\texttt{==},\texttt{!=},\texttt{!},\texttt{\&},\texttt{|}\} \cup \{C\ \texttt{isclassof} \mid C \in \mathsf{ClassId}\} & & \\
lop \in \{\texttt{*},\texttt{-*},\texttt{\&}\} & qt \in \{\texttt{ex},\texttt{fa}\} & \ell \in \mathsf{RdWrVar} & \\
e \in \mathsf{Exp} & ::= & \pi \mid \ell \mid op(\bar{e}) & \\
F \in \mathsf{Formula} & ::= & e \mid v.f \overset{\pi}{\mapsto} e \mid v.\kappa\texttt{<}\bar{\pi}\texttt{>} \mid F\ lop\ F \mid (qt\ T\ x)(F) & \\
\textit{Derived forms:} & & v.f \overset{\pi}{\mapsto} T \triangleq (\texttt{ex}\ T\ x)(v.f \overset{\pi}{\mapsto} x) & \\
& & v.\kappa\texttt{<}\bar{\pi},T,\bar{\pi}'\texttt{>} \triangleq (\texttt{ex}\ T\ x)(v.\kappa\texttt{<}\bar{\pi},x,\bar{\pi}'\texttt{>}) & \\
& & F\ \texttt{*-*}\ G \triangleq (F\ \texttt{-*}\ G)\ \texttt{\&}\ (G\ \texttt{-*}\ F) & \\
& & F\ \texttt{ispartof}\ G \triangleq G\ \texttt{-*}\ (F\ \texttt{*}\ (F\ \texttt{-*}\ G)) &
\end{array}$$

Appendix A presents a natural deduction calculus for our logical operators. The natural deduction rules are the standard rules of (affine) linear logic (Wadler 1993). They are also the standard rules of (affine) BI (O'Hearn and Pym 1999), 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 pretty standard and is detailed in our technical report (Haack and Hurlin 2008a). There we also prove the soundness of our 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`.

We assume that the `Object` class contains the following default declaration of the `space`-predicate:

```
class Object {
    //@ pred space<perm p> = true;
    //@ axiom space<p> *-* (space<p/2> * space<p/2>);
}
```

In the class 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 `space`-predicate in the `Object` class is meant to be extended in subclasses. Note that the class axiom imposes a constraint on the way subclasses may extend it.

```
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. Mutable and immutable integer objects

Figure 2 presents classes for mutable and immutable integer objects, which we will use in examples throughout this paper. We explicitly quantify over auxiliary variables in method contracts, enclosing quantifiers in angle brackets in front of method declarations (analogously to type parameters in Java). Note that the `space`-predicate in the `Integer` class ignores its parameter. This is intentional. As a result, the class axiom for `Integer`s trivially holds.

3. Iterator Protocols

3.1 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();
}
```

The interface can be instantiated to give rise to read-write iterators, read-only iterators, and iterators over collections of immutable objects, see Figure 3. In the third instantiation, direct access of immutable collection elements is allowed. In the first two instantiations, access to mutable collection elements is governed by the collection. Fortunately, however, the collection releases access control to collection elements that get removed, see the read-write iterator in Figure 3.

Read-write iterator (Protocol 1):

```
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> }
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 (Protocol 1):

```
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 (Protocol 1):

```
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> }
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)
```

A shallow iterator (Protocol 2):

```
Collection<false> c = new List();
    { c.space<1> }
MapEntry e = new MapEntry(0,42);
    { c.space<1> * e.space<1> * e.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 ( c.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 3. Instances of Protocols 1 and 2

```
final class MapEntry {
    private int key;
    private int val;
    //@ pred ispace = (ex perm p)(key |-p-> int);
    //@ pred space<perm p> = val |-p-> 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. A map entry class

3.2 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 rights 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 the element state—for instance the key in case of map entries. To safely share element state between the collection and its client, the shared state (e.g., the key) has to be immutable. We therefore introduce a second generic predicate `ispace` to represent the immutable part of an object space:

```
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);
}
```

Figure 4 shows a `MapEntry` class, where the `key`-field constitutes the immutable part and the `val`-field the mutable part.

In order to have a uniform type for deep and shallow collections, we parametrize the collection type by a boolean flag:

```
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();
}

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();
}
```

When we instantiate the boolean flag `isdeep` by `true`, we obtain a deep collection with a protocol that is essentially equivalent to Protocol 1. Instantiating `isdeep` by `false` results in a shallow collection where the immutable parts of the elements are freely shared between collection and elements, and access rights to the mutable element parts remain with the elements. Figure 3 instantiates the generic collection interface to a shallow collection.

3.3 Protocol 3 — Permission-parametrized Iterator States

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

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

This axioms allows to convert a read-only iterator to a read-write iterator when other concurrent read-only iterators have terminated, see Figure 5. 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();
}
```

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.

```
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) it.next();
    MutableInteger x2 = (MutableInteger) it.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 5. An Instance of Protocol 3

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 the proof for Protocol 1, but heavier in notation. In contrast to Protocols 1 and 3, the implementation for Protocol 2 has to parametrize the `Node` class by a boolean flag `isdeep`.

Protocols 1 and 3 make use of the following `Node` class:

```
final class Node {

    /*@ spec_public @*/ private Object val;
    /*@ spec_public @*/ private 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; }
}
```

This class makes use of the `spec_public` modifier for fields and predicates, which is syntactic sugar:

- Declaring a (possibly private) 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:

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

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

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

In the definition of the `space`-predicate of `Node`, we use helper predicates `valspace<`p,x`>` and `nextspace<`p,x`>`. These predicates are defined in Figure 6, 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. The following abbreviation is also handy:

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

To technically simplify our proofs, we have declared the `Node` class `final`. This is not essential. If `Node` were not `final`, we would have to augment our helper predicates in Figure 6 by additional formulas of the shape $\mathit{rest}(p,x,\texttt{Node})$, in order to represent the part of x's heap space that is accounted for in x's dynamic class, but not in the `Node` class (which is a superclass of x's dynamic class).

LEMMA 1.

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

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`>`

4.1 Implementing Interface 1

Figure 7 shows an implementation of our iterator interface for Protocol 1. Note that we have declared the `ListIterator` class `final`. Consequently, predicates of the form v`.`P`@ListIterator<`$\bar{\pi}$`>` are equivalent to $v.P\texttt{<}\bar{\pi}\texttt{>}$. 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 8 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 @*/ private Node header;
    //@ spec_public extends group space<perm p> by
    //@   (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>@*/
{
    private Node cur, prev, pprev;
    //@ pred ready =  pprev<1,Node> * (ex Node y,z)(
    //@   prev<1,y> * cur<1,z> * tail1<p,y,z> *
    //@   diff<p,iteratee,y>);
    //@ pred readyForNext =  pprev<1,Node> * (ex Node y,z)(
    //@   prev<1,y> * cur<1,z> * tail2<p,y,z> *
    //@   diff<p,iteratee,y>);
    //@ pred readyForRemove<Object e> =  (ex Node x,y,z)(
    //@   pprev(1,x) * prev(1,y) * prev (1,z) *
    //@   tail4<p,x,y,z,e> * diff<p,iteratee,x>);
    //@ req iteratee.space<p> * list==iteratee; ens ready<p>;
    ListIterator(List list) {
        prev = list.getHeader();
        cur = prev.getNext();
    }
    //@ req ready; ens ready & (result -* readyForNext);
    public boolean hasNext() {
        return cur != null;
    }
    //@ req readyForNext;
    //@ ens readyForRemove<result> * result.space<p>;
    public Object next() {
        pprev = prev;
        prev = cur;
        cur = cur.getNext();
        return prev.getVal();
    }
    //@ req readyForRemove<Object> * p==1; ens ready;
    public void remove() {
        pprev.setNext(cur);
        prev = pprev;
    }
}
```

Figure 7. An implementation for Protocol 1

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 part (b) what is needed to prove the "`readyForRemove`-to-`ready`" axiom.

LEMMA 4.

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

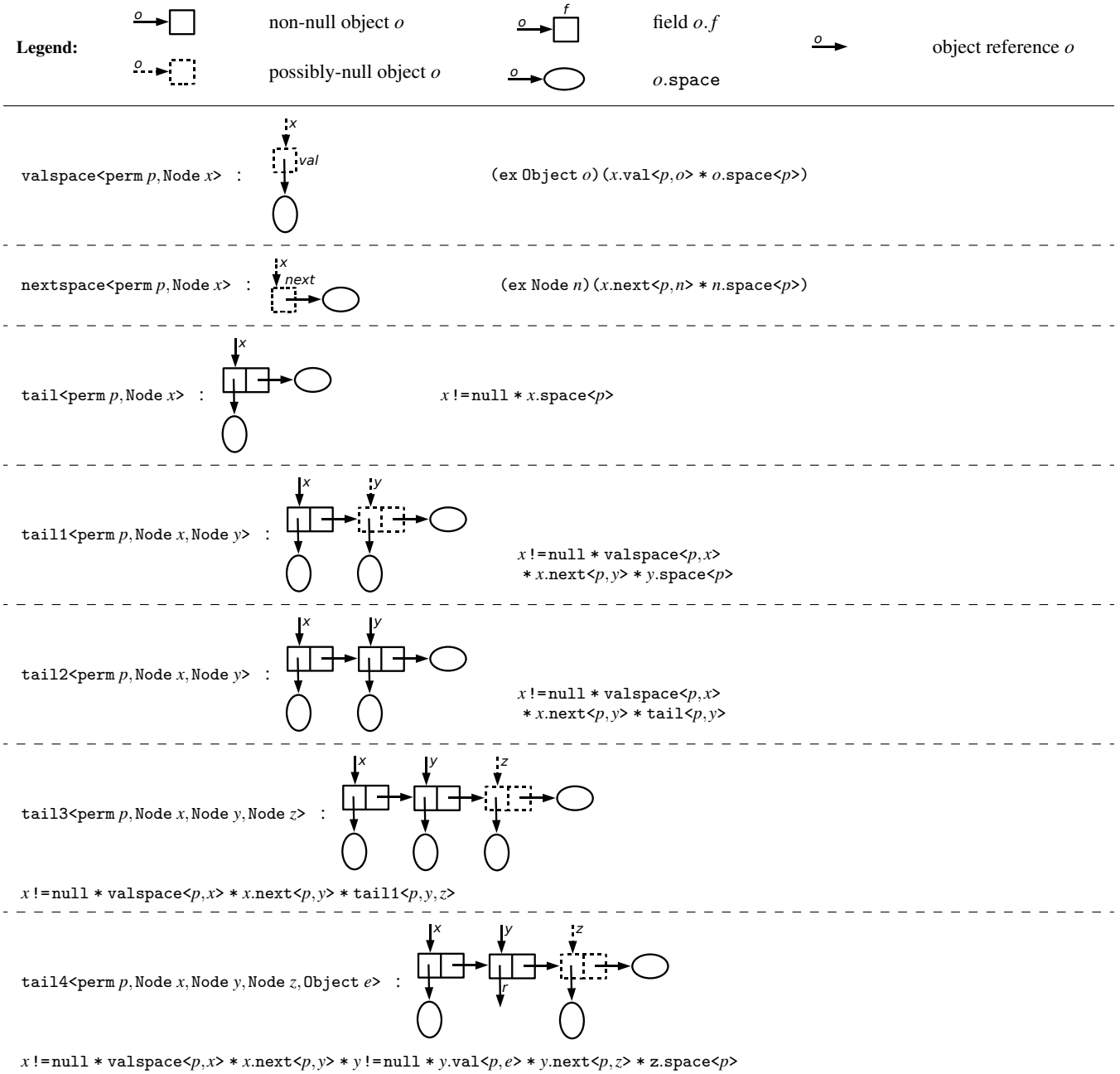

Figure 6. Helper predicates

Expanded method body with pre/postcondition:

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

Proof outline:

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

Figure 8. Proof outline for `next()`

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:

$$\texttt{tail3<}p,x,y,z\texttt{>} \mathrel{-\!\!*} (\texttt{valspace<}p,x\texttt{>} * x\texttt{.next<}p,y\texttt{>} * \texttt{tail1<}p,y,z\texttt{>}) \quad (1)$$

$$(\texttt{valspace<}p,x\texttt{>} * x\texttt{.next<}p,y\texttt{>} * y\texttt{.space<}p\texttt{>}) \mathrel{-\!\!*} x\texttt{.space<}p\texttt{>} \quad (2)$$

The following formulas can be verified by natural deduction:

$$A * (B \mathrel{-\!\!*} C) * (A \mathrel{-\!\!*} A') * (B' \mathrel{-\!\!*} B) \mathrel{-\!\!*} A' * (B' \mathrel{-\!\!*} C) \quad (3)$$

$$A * D * (D * B \mathrel{-\!\!*} C) \mathrel{-\!\!*} A * (B \mathrel{-\!\!*} C) \quad (4)$$

Now suppose that:

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

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

$$\texttt{valspace<}p,x\texttt{>} * x\texttt{.next<}p,y\texttt{>} * \texttt{tail1<}p,y,z\texttt{>} * (\texttt{valspace<}p,x\texttt{>} * x\texttt{.next<}p,y\texttt{>} * y\texttt{.space<}p\texttt{>} \mathrel{-\!\!*} c\texttt{.space<}p\texttt{>})$$

Applying (4), we then obtain:

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

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

4.2 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 of Protocol 3:

$$\texttt{iteratee.space<1/2> * ready<1/2> -* ready<1>} \quad (5)$$

To this end, we define the following auxiliary combinators:

$$\mathit{double}(P) \triangleq P * P$$

$$\mathit{bump}(p,c,x) \triangleq p\texttt{!=1} * c\texttt{.space<}p\texttt{>} \mathrel{-\!\!*} \mathit{double}(\texttt{tail<}p,x\texttt{>} * \texttt{diff<}p,c,x\texttt{>})$$

$$\mathit{maybump}(P,p,c,x) \triangleq (P * \texttt{diff<}p,c,x\texttt{>}) \;\&\; \mathit{bump}(p,c,x)$$

Intuitively, the `maybump`-combinator provides the choice to either keep iterating normally (first factor), or else pay $c\texttt{.space<}p\texttt{>}$ in order 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 cur, prev, pprev;
    //@ pred ready<perm p> =  pprev<1,Node> * (ex Node y,z)(
    //@   prev<1,y> * cur<1,z> *
    //@   maybump(tail1<p,y,z>, p, iteratee, y) );
    //@ pred readyForNext<perm p> =  pprev<1,Node> * (ex Node y,z)(
    //@   prev<1,y> * cur<1,z> *
    //@   maybump(tail2<p,y,z>, p, iteratee, y) );
    //@ pred readyForRemove<perm p, Object e> =  (ex Node x,y,z)(
    //@   pprev(1,x) * prev(1,y) * prev (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 the following lemma:

LEMMA 5. $c\texttt{.space<}\frac{1}{2}\texttt{>} * \mathit{maybump}(\texttt{tail1<}\frac{1}{2},y,z\texttt{>}, \frac{1}{2}, c, y) \mathrel{-\!\!*} \mathit{maybump}(\texttt{tail1<1},y,z\texttt{>}, 1, c, y)$

Proof. By natural deduction, using axiom (Ax Share). □

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 6.

(a) $c\texttt{.header<}p,h\texttt{>} * \texttt{tail<}h,p\texttt{>} * \mathit{rest}(p,c,\texttt{List}) \mathrel{-\!\!*} \mathit{bump}(p,c,h)$

(b) $\mathit{maybump}(\texttt{tail3<}p,x,y,z\texttt{>}, p, c, x) \mathrel{-\!\!*} \mathit{maybump}(\texttt{tail1<}p,y,z\texttt{>}, p, c, y)$

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

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

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:

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

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

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

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 (Parkinson 2005), higher-order separation logic (Krishnaswami 2006), a linear typestate system (Bierhoff 2006; Bierhoff and Aldrich 2007), and a linear type-and-effect system (Boyland et al. 2007).

Parkinson (Parkinson 2005) uses iterators as an example. He supports simultaneous read-only iterators through counting permissions, rather than fractional permissions. His iterators are shallow, and do not give iterator clients access to collection elements that are retrieved by the iterator.

(Krishnaswami 2006) uses higher-order predicates to specify a protocol for iterators over linked lists in higher-order separation logic. His iterators are read-only and shallow. His protocol allows multiple active iterators over the same collection and enforces that they are all abandoned once an element is added to the collection. The protocol uses a single linear token to pass the access right from iterator A to iterator B (and possibly later back to A via the same token). In this sense, multiple iterators have to "synchronize". In particular, in concurrent programs multiple read-only iterators are not supported without synchronization, as facilitated by fractional or counting permissions.

(Bierhoff and Aldrich 2007) present a linear typestate system based on a fragment of linear logic. The fragment includes multiplicative conjunction, additive conjunction and additive disjunction in pre- and postconditions, and uses $\multimap$ as a separator between pre- and postconditions. In (Bierhoff and Aldrich 2007) and (Bierhoff 2006), the authors use iterators as a case study for their typestate system. Because their system has fractional permissions, they support concurrent read-only iterators. Their protocols do not not support deep iterators over *mutable* collection elements, although (Bierhoff 2006) 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, while still staying in the decidable MALL fragment of linear logic. 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 their papers presents an iterator implementation, or a mapping of iterator state predicates to concrete definitions.

Boyland, Retert and Zhao (Boyland et al. 2007) informally explain how to apply their linear type and effect system (an extension of (Boyland and Retert 2005) with fractional permissions) to specify and verify iterator protocols. Their system facilitates concurrent read-only iterators through fractional permissions. The paper does not address deep iterators. In contrast to our `Iterator` interface, (Boyland et al. 2007)'s interface is not parametrized by the collection. As a result, our methods will sometimes need auxiliary parameters (in angle brackets), where theirs do not. Like us, (Boyland et al. 2007) 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 has a different semantics than separation logic's magic wand (which we use).

Compared to the related work discussed above, we believe our main technical contribution is support for *deep iterators*. None of the protocols above allows read/write access to collection elements that get returned by `next()`. Our protocols grant read/write access to these 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.

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 (Bierhoff 2006; Bierhoff and Aldrich 2007), 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 (Krishnaswami 2006)). This is not surprising, because implementing linked data structures is error prone and certainly much harder than using iterators in a disciplined way. We note, however, that 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.

Acknowledgments

We thank Marieke Huisman and Erik Poll for interesting discussions about this work, and the anonymous reviewers for useful comments.

References

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.

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

K. Bierhoff and J. Aldrich. Modular typestate verification of aliased objects. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, 2007.

R. Bornat, P. O'Hearn, C. Calcagno, and M. Parkinson. Permission accounting in separation logic. In *Principles of Programming Languages*, pages 259–270, New York, NY, USA, 2005. ACM Press. ISBN 1-58113-830-X.

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.

J. Boyland and W. Retert. Connecting effects and uniqueness with adoption. In *Principles of Programming Languages*, 2005.

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.

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

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

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

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.

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

C. Haack and C. Hurlin. Separation logic contracts for a Java-like language with fork/join. In *Algebraic Methodology and Software Technology*, number 5140 in Lecture Notes in Computer Science, pages 199–215. Springer-Verlag, 2008b.

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

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

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

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

M. Parkinson and G. Bierman. Separation logic and abstraction. In *Principles of Programming Languages*, 2005.

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, *CARDIS 2004*. Kluwer Academic Publishing, 2004.

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

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.

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

A. Verification Rules

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

(Id) $\dfrac{\Gamma \vdash \bar{F}, G : \diamond}{\Gamma;\bar{F}, G \vdash G}$ (Ax) $\dfrac{\Gamma \vdash F \quad \Gamma \vdash F : \diamond}{\Gamma;\ \vdash F}$ (∗ Intro) $\dfrac{\Gamma;\bar{F} \vdash H_1 \quad \Gamma;\bar{G} \vdash H_2}{\Gamma;\bar{F},\bar{G} \vdash H_1 * H_2}$ (∗ Elim) $\dfrac{\Gamma;\bar{F} \vdash G_1 * G_2 \quad \Gamma;\bar{E}, G_1, G_2 \vdash H}{\Gamma;\bar{F},\bar{E} \vdash H}$ (-∗ Intro) $\dfrac{\Gamma;\bar{F}, G_1 \vdash G_2}{\Gamma;\bar{F} \vdash G_1 \mathbin{-\!\!*} G_2}$

(-∗ Elim) $\dfrac{\Gamma;\bar{F} \vdash H_1 \mathbin{-\!\!*} H_2 \quad \Gamma;\bar{G} \vdash H_1}{\Gamma;\bar{F},\bar{G} \vdash H_2}$ (& Intro) $\dfrac{\Gamma;\bar{F} \vdash G_1 \quad \Gamma;\bar{F} \vdash G_2}{\Gamma;\bar{F} \vdash G_1 \,\&\, G_2}$ (& Elim 1) $\dfrac{\Gamma;\bar{F} \vdash G_1 \,\&\, G_2}{\Gamma;\bar{F} \vdash G_1}$ (& Elim 2) $\dfrac{\Gamma;\bar{F} \vdash G_1 \,\&\, G_2}{\Gamma;\bar{F} \vdash G_2}$ (Fa Intro) $\dfrac{x \notin \bar{F} \quad \Gamma, x : T;\bar{F} \vdash G}{\Gamma;\bar{F} \vdash \texttt{(fa}\,T\,x\texttt{)(}G\texttt{)}}$

(Fa Elim) $\dfrac{\Gamma;\bar{F} \vdash \texttt{(fa}\,T\,x\texttt{)(}G\texttt{)} \quad \Gamma \vdash \pi : T}{\Gamma;\bar{F} \vdash G[\pi/x]}$ (Ex Intro) $\dfrac{\Gamma, x : T \vdash G : \diamond \quad \Gamma \vdash \pi : T \quad \Gamma;\bar{F} \vdash G[\pi/x]}{\Gamma;\bar{F} \vdash \texttt{(ex}\,T\,x\texttt{)(}G\texttt{)}}$ (Ex Elim) $x \notin \bar{F}, H$ $\dfrac{\Gamma;\bar{E} \vdash \texttt{(ex}\,T\,x\texttt{)(}G\texttt{)} \quad \Gamma, x : T;\bar{F}, G \vdash H}{\Gamma;\bar{E},\bar{F} \vdash H}$

Java Axioms, $\Gamma \vdash F$:

(Ax True) $\dfrac{}{\Gamma \vdash \texttt{true}}$ (Ax False) $\dfrac{}{\Gamma \vdash \texttt{false} \mathbin{-\!\!*} F}$ (Ax Pure) $\dfrac{}{\Gamma \vdash (e \,\&\, F) \mathbin{-\!\!*} (e * F)}$ (Ax Subst) $\dfrac{\Gamma \vdash e, e' : T \quad \Gamma, x : T \vdash F : \diamond}{\Gamma \vdash (F[e/x] * e == e') \mathbin{-\!\!*} F[e'/x]}$ (Ax Bool) $\dfrac{\Gamma \models\, !e_1 \mid !e_2 \mid e'}{\Gamma \vdash (e_1 * e_2) \mathbin{-\!\!*} e'}$

(Ax Split/Merge) $\dfrac{}{\Gamma \vdash v.f \overset{\pi}{\mapsto} e \mathbin{*\!\!-\!\!*} (v.f \overset{\pi/2}{\longmapsto} e * v.f \overset{\pi/2}{\longmapsto} e)}$ (Ax Cl) $\dfrac{\Gamma \vdash \pi : t\texttt{<}\bar{\pi}'\texttt{>} \quad \mathsf{axiom}(t\texttt{<}\bar{\pi}'\texttt{>}) = F}{\Gamma \vdash F[\pi/\texttt{this}]}$ (Ax Open/Close) $\dfrac{\Gamma \vdash \texttt{this} : C\texttt{<}\bar{\pi}'\texttt{>} \quad \mathsf{pbody}(P\texttt{<}\bar{\pi}\texttt{>}, C\texttt{<}\bar{\pi}'\texttt{>}) = F \quad C\texttt{<}\bar{\pi}'\texttt{>}\ \texttt{extends}\ D\texttt{<}\bar{\pi}''\texttt{>}}{\Gamma \vdash \texttt{this}.P\texttt{@}C\texttt{<}\bar{\pi}\texttt{>} \mathbin{*\!\!-\!\!*} (F * \texttt{this}.P\texttt{@}D\texttt{<}\bar{\pi}\texttt{>})}$

(Ax Final) $\dfrac{\Gamma \vdash v : C\texttt{<}\bar{\pi}\texttt{>} \quad C \text{ is } \texttt{final}}{\Gamma \vdash C\ \texttt{isclassof}\ v}$ (Ax Null) $\dfrac{}{\Gamma \vdash \texttt{null.}\kappa\texttt{<}\bar{\pi}\texttt{>}}$ (Ax Sub Cl) $\dfrac{C \preceq D}{\Gamma \vdash v.P\texttt{@}D\texttt{<}\bar{\pi}\texttt{>}\ \texttt{ispartof}\ v.P\texttt{@}C\texttt{<}\bar{\pi}\texttt{>}}$ (Ax Sub Dyn) $\dfrac{}{\Gamma \vdash v.P\texttt{@}C\texttt{<}\bar{\pi}\texttt{>}\ \texttt{ispartof}\ v.P\texttt{<}\bar{\pi}\texttt{>}}$

(Ax Dyn) $\dfrac{}{\Gamma \vdash (v.P\texttt{@}C\texttt{<}\bar{\pi}\texttt{>} * C\ \texttt{isclassof}\ v) \mathbin{-\!\!*} v.P\texttt{<}\bar{\pi}\texttt{>}}$ (Ax Share) $\dfrac{\text{the hole in } F[\,] \text{ is not to the left of a } \mathbin{-\!\!*}}{\Gamma \vdash (\ v.f \overset{\pi}{\mapsto} e\ \ \&\ \ F[\texttt{(ex}\,T\,x\texttt{)(}v.f \overset{\pi'}{\mapsto} x * G\texttt{)}]\ ) \mathbin{-\!\!*} F[v.f \overset{\pi'}{\mapsto} e * G[e/x]]}$

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

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):

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

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

(Seq) $\dfrac{\Gamma \vdash \{F\}hc\{H\} \dashv \Gamma' \quad \Gamma' \vdash \{H\}c : T\{G\}}{\Gamma \vdash \{F\}hc; c : T\{G\}}$ (Return) $\dfrac{\Gamma \vdash v : T \quad \Gamma, \texttt{result} : T \vdash G : \diamond}{\Gamma \vdash \{G[v/\texttt{result}]\}v : T\{G\}}$ (Frame) $\dfrac{\Gamma \vdash H : \diamond \quad \Gamma \vdash \{F\}hc\{G\} \dashv \Gamma' \quad \mathsf{RdWrVar}(H) \cap \mathsf{Modifies}(hc) = \emptyset}{\Gamma \vdash \{F * H\}hc\{G * H\} \dashv \Gamma'}$

(Con) $\dfrac{\Gamma;F \vdash F' \quad \Gamma \vdash \{F'\}hc\{G'\} \dashv \Gamma' \quad \Gamma';G' \vdash G}{\Gamma \vdash \{F\}hc\{G\} \dashv \Gamma'}$ (Aux Var) $\dfrac{\Gamma, x : T \vdash \{F\}hc\{G\} \dashv \Gamma', x : T}{\Gamma \vdash \{\texttt{(ex}\,T\,x\texttt{)(}F\texttt{)}\}hc\{\texttt{(ex}\,T\,x\texttt{)(}G\texttt{)}\} \dashv \Gamma'}$ (Var Dcl) $\dfrac{\Gamma \vdash T : \diamond}{\Gamma \vdash \{\texttt{true}\}T\,\ell\{\ell == \mathsf{default}(T)\} \dashv \Gamma, \ell : T}$

(Get Var) $\dfrac{\Gamma \vdash \ell : T}{\Gamma \vdash \{\texttt{true}\}x\texttt{=}\ell\{x == \ell\} \dashv \Gamma, x : T}$ (Set Var) $\dfrac{\Gamma \vdash v : \Gamma(\ell)}{\Gamma \vdash \{\texttt{true}\}\ell\texttt{=}v\{\ell == v\} \dashv \Gamma}$ (Op) $\dfrac{\Gamma \vdash op(\bar{v}) : T}{\Gamma \vdash \{\texttt{true}\}x\texttt{=}op\texttt{(}\bar{v}\texttt{)}\{x == op(\bar{v})\} \dashv \Gamma, x : T}$

(Get) $\dfrac{\Gamma;F \vdash v.f \overset{\pi}{\mapsto} w \quad \Gamma \vdash v : V \quad W\,f \in \mathsf{fld}(V)}{\Gamma \vdash \{F\}x\texttt{=}v.f\{F * x == w\} \dashv \Gamma, x : W[v/\texttt{this}]}$ (Set) $\dfrac{\Gamma \vdash v : V \quad W\,f \in \mathsf{fld}(V) \quad \Gamma \vdash w : W[w/\texttt{this}]}{\Gamma \vdash \{v.f \overset{1}{\mapsto} W\}v.f\texttt{=}w\{v.f \overset{1}{\mapsto} w\} \dashv \Gamma}$

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

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

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