Iterators can be Independent "from" Their Collections

John Boyland
Nanjing University, China
University of Wisconsin-Milkaukee, USA
boyland@cs.uwm.edu

William Retert Yang Zhao
University of Wisconsin-Milwaukee, USA
williamr@cs.uwm.edu yangzhao@cs.uwm.edu

Abstract

External iterators pose problems for alias control mechanisms: they have access to collection interals and yet are not accessible from the collection; they may be used in contexts that are unaware of the collection. And yet iterators can benefit from alias control because iterators may fail "unexpectedly" when their collections are modified. We explain a novel aliasing annotation "from" that indicates when a collection intends to delegate its access to internals to a new object and how it can be given semantics using a fractional permission system. We sketch how a static analysis using permissions can statically detect possible concurrent modification exceptions.

1. Introduction

Iterators in Java™and related languages are objects that give sequential access to collections, while being external to the collection. In particular, multiple iterators can operate on a collection at the same time, and the collection is not directly involved in the operation of iterators. Iterators are an improvement over previous related concepts, such as *cursors*, precisely because of this independence. Typically a collection only had one cursor, and moving the cursor had an effect on the collection. Multiple cursors are possible but hard to manage. Noble [18] has surveyed a wide variety of iterator architectures; we focus here on "external iterators."

An iterator is originally created by the collection, but after creation, it may be used in contexts in which the collection is neither visible nor in scope. The very independence that makes iterators so powerful also makes programs that use them more complex because of the interactions, notably aliasing, between the collection and the iterators. In particular, an iterator typically has pointers into the internals of the collection representation, and may even perform changes on this representation.

Figure 1 gives two interface declarations for the kinds of iterators that will be discussed in this paper. In Java, these interfaces are conflated by making `remove` an optional operation. In this work, to make it easier to reason about iterators and to make the distinction visible in the type system, we use separate interfaces. C++ similarly makes a type-level distinction between iterators that can be used to modify a collection and those that cannot. Other kinds of iterators (such as `ListIterators` that can change an element in a collection) could be defined. We will use the term *mutating iterator* to refer to generally to any iterator that can modify the underlying collection.

Additionally, we have annotated the iterator methods with "method effects" indicating what state the methods are intended or permitted to access. We will assume that all methods are so annotated. In this case, the methods are declared as accessing only the state of the iterator and not reading or writing any other state. The fact that we can mask away the effects on the collection is non-trivial and is one of the main discussion topics of this paper.

```
interface Iterator {
  reads(this.All)
  boolean hasNext();

  writes(this.All)
  Object next();
}
```

```
interface RIterator
  extends Iterator {

  writes(this.All)
  void remove();
}
```

Figure 1. Two classes of iterators.

```
class App {
  this List list = new List();
  ⋮
  writes(All) void run() {
    Iterator it = list.iterator();
    ⋮ What if we mutate list?
    if (Util.member(null,it)) { ... }
  }
}
```

Figure 2. Using an iterator.

Normally, we will follow standard OO convention and abbreviate `this.All` as `All`, since `All` is a (model) instance field.

Figure 2 shows an example of using iterators. It uses classes `List` and `Util` defined in the following section. The example includes some omitted code that may or may not mutate the collection. A collection may be changed directly using a method such as `clear` or `add`, as well as indirectly through a mutating iterator. When this happens, all iterators currently active on the collection (excepting only the iterator through which the mutation took place, if any) are potentially invalid: they may be referring to internals that have been discarded or are otherwise reorganized. For instance, if a linked list is cleared, then existing iterators may refer to nodes that are (otherwise) garbage, and indeed in a language such as C++, the node may have already been returned to the memory allocation system. If a new entry is added to a hash table, an existing iterator may find its node pointer has been rehashed to a new location and continuing the iteration may result in repeating some elements previously encountered, and omitting others. In C++, the programmer is warned by the documentation that existing iterators may be "invalid" after a mutation.

It is possible to implement iterators so that they are robust in the face of change, albeit with some additional complexity. In Java, rather than following this route or letting a potentially confusing sit-

uation emerge, a *fail fast* semantics is used: an iterator will (almost) always detect when a mutation outside of its control has happened, and throw a `ConcurrentModificationException` (CME) if it is used afterwards. Typically this is implemented by version stamping the iterator and collection, but the implementation details are not the focus here. Instead we are interested in how static alias control mechanisms can (1) describe external iterators, (2) explain the effects of using a (mutating) iterator, (3) explain why and when concurrent modification exceptions are thrown, and (4) statically prevent these exceptions from occurring.

In the following section, we look at a linked list class with iterators annotated to express the design intent of the aliasing. Then in Sects. 3 and 4, we look at previously described alias control systems and evaluate them by these four criteria. In Section 5, we describe how the design intent of the example can be expressed using our "fractional permission" system.

2. Example

Figure 3 defines a simple linked list class with two kinds of iterators. The code is annotated (*italic* words) with "design intent" showing how aliases are intended to be controlled. Except for *from* (explained below), these annotations have appeared in one form or another in previous work. The example also uses class parameters (inside < >) to pass objects that may be used in annotations on fields of the class. For simplicity, we don't make use of generic classes (as in Java 5), although it has been shown that one can fruitfully use both class parameters for ownership and for generic classes [19].

Every field, parameter or return value is annotated by an aliasing annotation: *shared* (owned by the global context), *name* (owned by *name*), *readonly-name* (read-only state owned by *name*), *from(...)* (explained below). Another possibility is *unique*, not used here. The default is *borrowed*, which can be applied only to parameters (including method receivers), which means the method can only access state from the parameter (receiver) if the state is present in the *method effect*. A method effect is of the form *reads(...)* or *writes(...)* and permits the method access the mutable state named; write access includes read access. Here *All* means all state of this object, or any object owned by it (transitively). Constructors are implicitly permitted to write any part of the constructed object's state.

For example, the `add` method of class `List` is annotated *writes(All)* which permits it to read or write any field of the list object (or its nodes, which it owns). Here it simply updates the `head` field. The parameter is *shared* which means there is no alias control intended here.

Next consider the `iterator()` method. Its effect *reads(All)* permits it to read any field or node of the list. The return value is annotated *from(All)* which means that the iterator is "unique" (unaliased with anything else) but that it gets (some of) its state from the method effect on *All*. The idea is that the collection temporarily *yields* its own state to the iterator, an independent (even unique) object. Indeed, the iterator becomes the owner of the (read-only) state of this list, as can be seen by the annotation on the parameter `list` of class `ListIter`.

We will continue with Figure 3, but first glance at Figure 4 which shows how code can use the iterator without reference to the collection. Methods `member` and `removeAll` both take iterators and are annotated that they modify the iterator (*writes(it.All)*) and nothing else.

Back at Figure 3, looking at class `ListIter`, one sees that field `cur` points to a node owned by list. The method `next()` is allowed to access the `cur.next` field because through its read-only ownership of the list, the iterator has read-only access to the internals.

```
class ListNode<owner> {
 owner ListNode<owner> next;
 shared Object datum;

 ListNode(shared Object d, owner Node n)
 { datum = d; next = n; }
}

class List {
  this ListNode<this> head;

  List() { head = null; }

  writes(All) void add(shared Object datum)
  { head = new ListNode<this>(datum,head); }

  writes(All) void clear() { head = null; }

  reads(All) from(All) Iterator iterator()
  { return new ListIter<this>(head); }

  writes(All) from(All) RIterator riterator()
  { return new ListRemoveIter<this>(this); }
}

class ListIter<readonly-this list>
    implements Iterator {
  list ListNode cur;
  Iterator(list ListNode head) { cur = head; }

  reads(All) boolean hasNext()
  { return cur!=null; }

  writes(All) shared Object next()
  { if (cur == null) return null;
    Object temp = cur.datum;
    cur = cur.next;
    return temp; }
}

class ListRemoveIter<this list>
    extends ListIter<list> implements RIterator {
  this List list; // MUST == class parameter list
  list ListNode prev, last;

  RIterator(this List list) // MUST == class param. list
  { super(list.head); this.list = list; }

  writes(All) shared Object next()
  { prev = last; last = cur;
    return super.next(); }

  writes(All) void remove()
  { if (prev == null) list.head = cur;
    else prev.next = cur;
    last = prev; }
}
```

Figure 3. Listed list with iterators annotated with design intent.

```
class Util {
  writes(it.All)
  static boolean member(Object x, Iterator it)
  { while (it.hasNext())
    { if (it.next() == x) return true; }
    return false; }

  writes(it.All)
  static void removeAll(Object x, RIterator it)
  { while (it.hasNext())
    { if (it.next() == x) it.remove(); } }
}
```

Figure 4. Independence of iterators.

```
class Iterator2 implements Iterator {
  this Iterator it1, it2;
  Iterator2(this Iterator it1,
            this Iterator it2) { ... }

  reads(All) boolean hasNext()
  { return it1.hasNext()||it2.hasNext();}

  writes(All) shared Object next()
  { return it1.hasNext() ?
           it1.next() : it2.next(); }
}
```

Figure 5. Constructing iterators using iterators.

The remove iterator `ListRemoveIter` extends the `ListIter` class with three more fields. The first field shares the same name as the class parameter and must indeed be the same (identical) pointer. The new fields are implicitly included in `All` and thus the overriding of `next()` is permitted to access the `prev` and `next` fields. More importantly `ListRemoveIter` requires that it be made (temporary) owner of the collection (for both read and write access). The `remove()` method uses this ownership to perform modifications to the list under the effect *writes(All)*.

What about the problems with concurrent modification? In Figure 2, the `run()` method gets an iterator and after some time uses it to check for nulls in the list. If there is an intervening modification, the call to `member` would "fail fast" with Java iterators. In this case, the design intent indicates that the iterator `it` has (temporarily) taken over read-only ownership of the list; if we permit a write of the list to happen, we are indeed permitting a write to occur concurrently with a read, a classic error.

According to the annotation, therefore, the list may not be mutated until the iterator is no longer in use. Dually, write access is permitted only at the cost of disallowing any later use of the iterator. With a mutating iterator, even read access is prohibited until when the mutating iterator is done, or dually, the mutating iterator may not be used once any (even read) access is performed.

One reason to permit the iterator to be used separately from the collection is to support existing code patterns. Examining the open-source Eclipse project for uses of iterators, we have found some cases of interest. One example (greatly simplified here) concerns building an iterator that is constructed from other iterators (see Figure 5). The compound iterator indirectly takes over the (temporary, read-only) ownership of the collections' internals (there may be more than one collection involved). Again this means that effects

```
class OtherCollection {
  this LinkNode<this> head;
  ⋮
  reads(All) from(All) Iterator iterator()
  { return new List(head).iterator(); }
}
```

Figure 6. Delegating iterator creation.

on the element iterators are mapped into effects on the compound iterator.

Figure 6 shows another pattern, whereby a class does not implement its own iterators and instead creates an instance of a collection and gets an iterator for it. (Here we assume a new constructor for `List` that takes a read-only list of nodes.) In the code we saw with this pattern, the delegation involved creating a `List` around an existing array.

In this section, we have showed several ways in which iterators are defined and used. Undoubtedly, the annotations and explanations here reflect our own biases (and we indeed show how they are realized in our "fractional permission" system), but the code itself is conventional. In the following sections, we survey previously described alias control systems and the extent to which they can describe what is going on in the code.

3. Ownership-based Alias Control

Ownership is a recognized alias control technique. With ownership, each object has another object as its *owner*. The root of the ownership hierarchy is often called "world." Clarke and others propose a owners-as-dominator model: any reference to an object must pass that object's owner [10, 11]. This encapsulation property prevents any access to an object from objects outside its owner, but rules out external iterators: If the iterator is totally outside of the representation of the collection object (Figure 7(a)), then the iterator is not able to access the internals. On the other hand, putting the iterator as part of the representation enables the references to the internals, but disables the references from outside of the collection (Figure 7(b)).

Since the iterator is a common idiom in OO programs, alternative models have been proposed. Clarke and Drossopoulou [9] permit iterator-like objects that violate owners-as-dominators to exist in stack variables but not to be stored in fields. Because they only have dynamic extent (rather than indefinite extent), these *dynamic aliases* are deemed less dangerous than "static aliases." However, the typing of these dynamic aliases requires that the collections be in scope. Therefore, the dynamic aliases solution cannot handle the iterator usages in Figs. 4, 5 and 6

A related relaxation of owners-as-dominators was formalized by Boyapati and others [4]. Here objects of inner classes are permitted to access the internals of the outer object, even though the inner class object is not necessarily owned by its outer object. An iterator implementation is declared as an inner class implementing a global interface. Again, the aliasing between the object and its inner class objects is deemed less hazardous since it is restricted to a single compilation unit. Originally, Boyapati permitted iterators to be passed outside of the scope of collections (as in Fig. 4), but in this case, the effects were imprecise: they operated on the "world." This extension proved unamenable to ownership-based checking of synchronization, and was dropped in subsequent work [3, page 36].

Ownership domains permit an owner to make some owned objects public while protecting others from external access [1]. The objects owned by an owner are partitioned into several "domains."

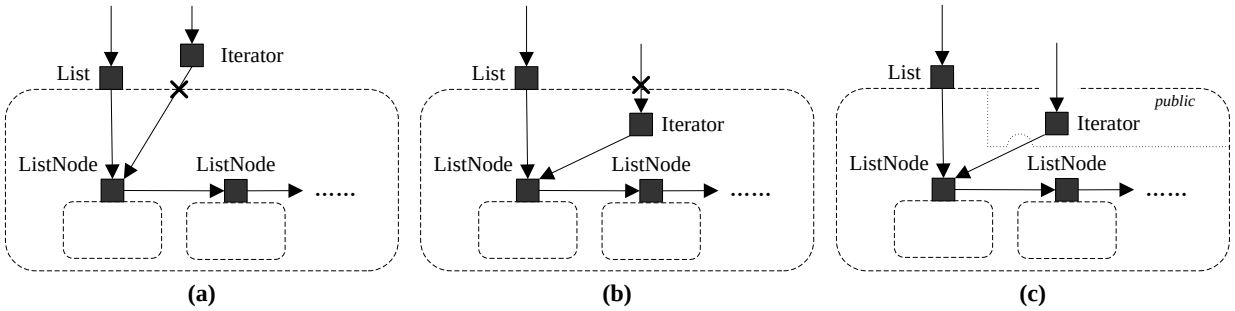


Figure 7. Owners-as-dominators (relaxed in (c)).

For instance, a collection object may own two domains: one for its internal representations and one for iterators. The latter domain can be public (see (c) in Figure 7). The iterator object can be referred to by outsiders (since it resides in a public domain), and it is able to access internal representations of the collection object (since its domain has the same owner as the representation domain). To precisely describe the states the iterator needs to access, Smith [20] proposes that the effects of an iterator be expressed as accessing state in its "sibling" domains, all other domains belonging to the same owner. This only works if the iterator is owned by the same object as the collection. Again there are problems with trying to use the iterator outside of the context of its collection's owner because then the sibling would be unknown.

Another alternate model is the "owners-as-modifier" model which distinguish between read-write and read-only references [13]. Read-write reference must pass through objects' owners, while read-only references may be created arbitrarly. The Universe type system distinguish three kinds of reference: *peer* references between objects in the same context; *rep* references from an object to any directly owned objects; *readonly* (or *any*) references between objects in arbitrary contexts. The last kind of references can not be used to modify objects. External iterators can be implemented in this model (see Figure 8), since any iterator object may use the readonly reference (represented as dashed line) to the internal representation of collections. Modifying iterators need to delegate mutations to the collection, which in turn must rely on runtime support to downcast the *any* reference to a *rep* reference.

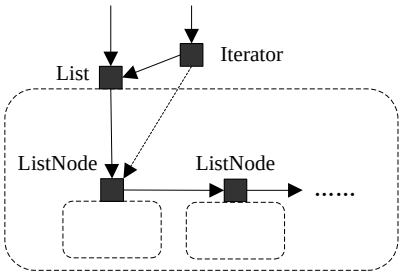


Figure 8. Owners-as-modifiers.

There are two difficulties, however. One is that object invariants cannot be guaranteed for objects referred to with "any" references. Thus a non-mutating iterator cannot rely on invariants of the collection (or its internals) to hold. Indeed, because of concurrent modifications, it is the case that a non-mutating iterator may fail unexpectedly. Furthermore, a mutating iterator must be a "peer" of the collection and thus cannot be used outside the context of the owner of the collection.

4. Handling Iterator Validity

Recently, a number of researchers tackled the problem of iterator validity, that is avoiding CME in Java-like languages. In particular, participants considered avoiding interference between calls that directly modify the collection and interleaved uses of one or more iterators to read that collection.

One solution, proposed by Weide [21] is to modify the semantics of the language such that the iterator copies out the contents of the collection upon its creation, and copies them back when it is finished. Changes made to the collection while the iterator is in use may be overwritten when the iterator is ended, using an explicit function in the collection. This behavior may be specified and checked using the standard tools of full program verification. These changes would affect neither asymptotic efficiency of iterator usage nor expressiveness when interference does occur; however, they are a significant departure from current usage.

The C# patterns for iterator usage and implementation are syntactically different from those of Java; in particular, enumerator functions can define iterators using `yield return` statements. Even so, the underlying problems of interference are generally the same: changes to the collection conflict with use of an iterator. Jacobs, Piessens, and Schulte [15] suggest defining *reads* clauses for enumerator methods that would declare some owned state as immutable while the enumerator-controlled loop is in effect, roughly correlating with the lifetime of the iterator in Java. This tracks well with C# enumerators' reading but not altering their collections; mutating iterators are not supported. Iterator objects can only be used directly to control loops, and may not be used independently of the collection. Every object is given special fields representing both overall writability and number of current readers. The reads clause is translated as modifications of and conditions on these fields, which may then be checked using the Boogie static verifier or using dynamic checks if necessary.

One may instead use fields to directly model the standard "timestamp" approach. Every collection is given an integer field which is incremented whenever the collection is modified; every iterator has an integer field with the value of the collection's timestamp at the time of the iterator's creation. David Cok [12] has instrumented this approach to iterators using model fields in JML. As these timestamps are only implemented in model fields, there is no concrete state underlying them. Rather than throw an exception when the collection's integer exceeds that its iterator, requirements on the relative values of the integers are included in the formal specification of the iterator. This specification can then be checked using ESC/Java2. In practice, the ESC/Java2 checker appears to

detect both legal and illegal uses by static inference; however, the correctness of this specification apparently cannot be proven.

Instead of specifying the values of a special fields as a signal of whether the collection has been or can be modified, one may directly encode modifications of the collection in an abstract predicate representing the collection as a whole. Krishnaswami [16] has done this using separation logic, whose predicates are comparable to permissions. Both the collection and the iterator get a high-level predicate that includes an abstract representation of the state of the collection. Most methods of the collection both require and return the predicate of the collection; methods that write the collection return it with a different abstract state. Creating a new iterator consumes the predicate for (permission to) the collection and produces that of the iterator. At any time, however, the predicate for the collection may be carved out of the predicate for its iterator, providing both the collection predicate and a "magic wand" implication that can consume the collection's predicate to recover the iterator. This implication essentially represents the non-collection portions of the iterator's state. The former may be used to call any of the collection's methods, including creating another iterator. Thus one may simultaneously have the predicate for the collection and any number of implications allowing one to exchange the collection predicate for some iterator. This must be done to use the iterator, after which the collection can be carved out once more. However, calling a method that modifies the collection returns a predicate with a different abstract state, which cannot be used to recover any of the current iterators–they are useless. Because this formalization lacks fractions [5, 8], even non-mutating iterators interfere with each other.

Bierhoff [2] describes another linear-based system using fractions and (linear) typestates. Typestates can represent sole, write, or read access to a program variable. Both the collection and the iterator have permissions defined for them; these also detail state changes in the objects themselves. For example, the `hasNext` method is needed to establish that the iterator's next is available as a prerequisite to calling `next`. The absence of this typestate precludes calling the method.

The iterator method returns a linear implication which consumes the permission for the collection, and produces only permission for the iterator. While the collection's permission is unavailable, methods that change it cannot be called. The `finalize` method of the iterator returns the reverse linear implication, consuming the iterator permission and producing that of the collection. The iterator class is parametrized by the collection to permit `finalize` to return permission to the collection. The iterator permission may be fractional, for a read-only iterator, or unique for a modifying iterator. The linear & is cleverly used to delay deciding whether an iterator is read-only or fractional. If the iterator permission is fractional, collection methods that only require fractional state may be called.

5. Explaining Iterators Using Permissions

The key issue with the design intent of the iterators is that access to the collection must be reduced to read-only (for non-mutating iterators) or completely prevented (for mutating iterators) while the iterator is active. In other words, the alias control system must be flow-sensitive. While some ownership type systems are moving to include flow-based analysis (see ownership transfer in Universes [17]), this is a significant increase in complexity.

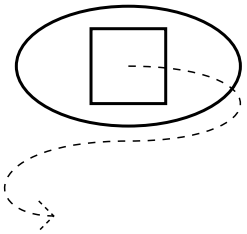
On the other hand, systems based on linear logic (such as Bierhoff's permission system or separation logic), lead to complex management of state. Linearity is powerful but is difficult to manage. Fähndrich and Deline [14] recognized this problem and designed a relation called "adoption" which permitted non-linearity to co-exist with linearity. We have since shown that adoption can model object ownership [6].

Our permission system combines the simplicity of ownership—allowing the iterator interface to hide the fact that it has access to the collection internals—with the power of linearity—expressing the constraint that the collection is encumbered by the iterator. In this section, we describe our permissions system and how it can account for the annotated design intent in the examples. Permissions are used to give a *semantics* to annotations, and then a flow-sensitive type system can be used to *check* that the code does indeed follow the design intent prescribed by the annotations.

5.1 Permissions overview

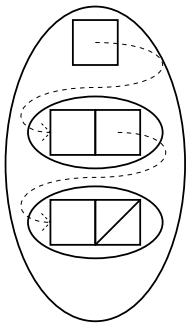
A *permission* is a token that permits access to mutable state. Each field in a Java program is associated with exactly one field permission. This permission can be *split* into *fractions*: in order to write a field one needs the whole field permission, but read access is permitted with only a non-zero fraction. Permissions cannot be copied—only transferred. As a result, although two read accesses can be carried out "at the same time" in different parts of the program, if one or both of the accesses is a write, the permission system will flag an error. This is the basic intuition of fractional permissions.

In order to support information hiding and "non-linear" access to state, we add the concept of *permission nesting* (a generalization of adoption); in which an arbitrary permission (often composite) is "nested" in a field permission. As a result, one who has access to the field (and knowledge of the nesting relationship) can get at the nested permissions, by "carving" them out of the field permission.

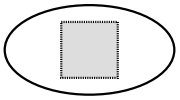


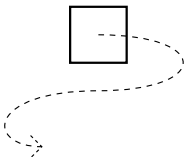
Nesting is used for two main purposes. On the left, we are using nesting to model "data groups." The square represents permission to access a field with a pointer value. The oval shows the (model) field "All." The digram demonstrates that the permission to access the field is nested in "All."

On the right, we are using nesting to express ownership as well as data groups. We now see that the field (small square near the top) points to a node whose entire state is nested in the first object's "All" mode field. This object has two fields (think of them as a data field and a next field), the second of which points to another owned node. The second node has a "null" next pointer.



Carving temporarily removes the permissions from the field permission; the field permission now has a "hole" in it. This situation is represented by a kind of "linear implication" $\Psi \multimap \Psi'$, where Ψ is the nested permission and Ψ' is the nester field permission.

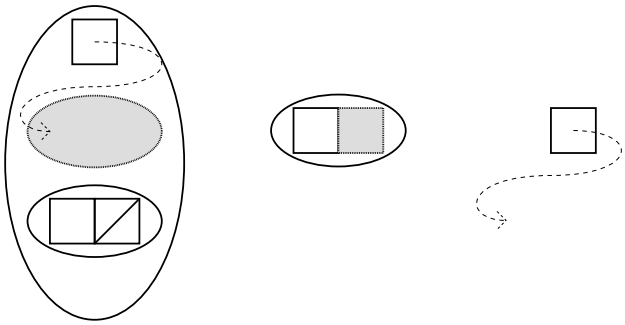




$(o.f \to r) \multimap o.\text{All}$ $\qquad$ $o.f \to r$

The permission $\Psi \multimap\!\!+ \Psi'$ can be read as referring to all the state implied by Ψ' except that part of which is implied by Ψ. In our permission system, "$\multimap\!\!+$" (read "scepter") functions very similarly to "$\rightharpoonup$" in separation logic (or "$\multimap$" in linear logic). The peculiar distinction of "$\multimap\!\!+$" is that it requires that the consequent (right-hand side) include the antecedent (left-hand side).

Carving is used to get any nested state. Thus to read the next field of the first node, we first carve out the node and then carve the field out:



The inversion of "carving" is "replacing." *Replacing* is simply linear *modus ponens*: it takes $\Psi \multimap\!\!+ \Psi'$ and Ψ and puts them together to retrieve the consequent Ψ'. In standard linear fashion, the process consumes both of the inputs (the antecedent and the implication).

Our permission logic also includes composition (represented by a comma), conditionals (to handle possibly null pointers) and existentials.

5.2 Annotation semantics

In current work [7], we describe the semantics of annotations precisely in terms of permissions. Here, we content ourselves with informal explanation:

data groups State encapsulation (such as `All` referring to all the state of an object) is handled by unit-typed fields that nest the state that belongs to it.

ownership Ownership is represented by nesting x`.All` of the owned object x in a field (ownership domain) of the owner. Multiple ownership domains [1] can be modelled. Every ownership domain is nested in `this.All`.

readonly ownership Read-only ownership is represented by nesting an unspecified *fraction* of the state of the owned object in the owner's domain.

borrowed Borrowed references are references that are transmitted without permissions; in order to access state through a "borrowed" parameter (or receiver), a method uses effects:

effects Method effects are represented by permissions that are passed to the method and which are returned afterwards.

unique A unique reference is always associated with permission to access the object it refers to (if any). For instance, a `unique` return value will be returned along with the necessary permissions.

from As with `unique`, a `from` return value has permissions to access it returned by the method, but unlike `unique`, these permissions encumber the effect named, the permissions for the effect are returned in linear implication:

$$(r.\text{All}) \multimap\!\!+ (\textit{effect}, v)$$

where v is an unspecified permission. In other words, the permissions represented by the effect are unusable until the state pointed to by the return value is no longer needed. At this time, the linear implication can be applied, releasing the effect and some unknown "residual" permission (v) that can be discarded. In our examples, the v is the iterator permission itself bereft of permission to access the collection.

$$\texttt{iterator}: \begin{array}{l}\forall zt \cdot (zt.\text{All} \rightarrow \\ \exists rv \cdot r.\text{All}, (r.\text{All} \multimap\!\!+ zt.\text{All}, v))\end{array}$$

Figure 9. The permission type of the `iterator()` method.

Annotations are translated into permissions using a simple substitution (not described here). An example is given in Fig. 9 and is explained below.

5.3 Controlling access

If one has permission to access an object's state, the carve operation gives access to the state nested in that object (other objects owned by it). Allowing this situation in general would break encapsulation; indeed it would permit clients to mutate list internals, resulting in complete chaos. The difficulty this situation presents cannot be solved simply by permissions because one still wishes to permit the client to call methods that use the permission to access internals. Instead, protection of internals is solved in the traditional way with visibility: carving is only permitted to access *visible state*.

The iterator examples show the list iterators accessing internals of the list, including its list nodes. In order to allow iterators to do these operations, we use Java's nested classes. The `ListIter` and `ListRemoveIter` classes are made (private) nested classes of `List` and thus given access to the list internals. The `ListNode` class serves as a structure and can make its fields public.

5.4 Explaining Iterators

In the `iterator()` method of class `List`, the read-only permission to access the list is passed to the new `ListIter` object so that it can nest this permission in its ownership domain. The permission for the effect is not returned to the caller as normal; a linear implication $\textit{Iter} \multimap\!\!+ (\texttt{reads(All)}, v)$ is returned instead. The read permission in the consequent cannot be used, as a read permission or to reform a write permission for the list, until the iterator is no longer in use, However, as fractions permit splitting a permission into an arbitrary number of read permissions, other read-only operations may be performed on the list. Figure 9 gives the full permission type of the method (after annotations are translated). The variables t and r refer to the receiver (`this`) and return value respectively. The variable z refers to the non-zero fraction of access required and v refers to the unspecified permission to be discarded when the linear implication is applied. The `riterator()` method is similar except that it requires, and so makes inaccessible, full (write) permission to the list.

The type in Fig. 9 does not disclose that the iterator uses ownership, or indeed anything in how the iterator is implemented. Indeed the permission implication $r.\text{All} \multimap\!\!+ (zt.\text{All}, v)$ is implemented internally by an empty permission since the return value's state $r.\text{All}$ includes the entire list read permission on the right-hand side of the implication. But the client does not need to know this; should *not* know this fact. The information hiding is essential to ensure that the list permission is inaccessible until the implication is applied to the iterator permission, consuming it and releasing the list permission.

In the iterator implementation, as the iterator is temporarily the owner of the list, it may be used without explicit mention of the list. In the implementation of the `next` method, we can access `cur.next` by first carving the (read-only) permission list from the iterator's "All" permission (provided by the caller per the method effect), and then getting access to the node by carving

its permission from the list, and finally to get access to the field by carving its permission from the node's "All" permission. This permission is existential (the next pointer is not determined by the node) and must be unpacked. At this point, we have a series of permission implications plus one field permission:

$$zl.\text{All} \multimap t.\text{All}, \qquad zc.\text{All} \multimap zl.\text{All},$$

$$(\exists p \cdot z(c.\texttt{next} \to p, \ldots)) \multimap zc.\text{All} \qquad zc.\texttt{next} \to n$$

Here t is the value of this, l for `list`, c for `cur` and `n` for its next field. When we are done, the entire set of permissions can be packed back up into t.All and returned to the caller. The caller need not be aware of the existence of the list at all.

For example, the `Util` methods work by borrowing iterators: its caller provides the permissions that are then returned. It is impossible to also provide permission to access the collection in a conflicting way—writing the collection is precluded by the presence of a read permission in the consequent of the linear implication–and thus call-back problems are prevented.

Remove iterators avoid throwing a CME because they nest the full write permission for the list preventing alteration of the list state until the remove iterator is no longer used. And, because creating iterators requires at least some part of the list's state, no other iterator can be created while the remove iterator is in use. Neither can a remove iterator be created if another iterator is in use. This is more strict than Java requires, but does ensure the absence of CMEs.

Regular iterators are also prevented from throwing a CME. In Fig 2, after the application requests for the iterator `it` to be created, the collection is encumbered. A linear implication for recovering the access to the list is made available. This linear implication can be applied at any time (in the static flow-analysis of the method). However, once it is applied, permission to access the iterator is irrevocably consumed. Thus, once the permission type checker is "forced" to apply the implication to permit a collection mutation, the iterator is no longer usable and later uses of the iterator will not type check.

The class to concatenate two iterators (see Fig. 5) requires ownership of the iterators, so that effects on them can be attributed to the compound iterator. This can be granted (the iterator was essentially "unique" before) at the price that the respective collections are still encumbered. In order to unencumber the collections, it necessary to retrieve the permissions for the iterators now nested in compound iterator; this can be done when the compound iterator is being discarded.

Delegation of iterator creation (see Fig. 6) uses an iterator on a newly constructed collection. The "*from*" annotation is not actually needed since the linked nodes are copied; it merely expresses the design intent that the collection should not be changed while the iterator is active.

5.5 Analysis

Our current work [7] gives a type system based on permissions. It is flow-sensitive, keeping track of the current permissions at each point in the program. When an field access is processed or a method is called, it checks whether the operation can be permitted. Nesting, carving and replacing operations are carried out implicitly as needed, perhaps several levels deep as the example for `cur.next` showed. The type system described is non-algorithmic, but we have a (more complex) algorithmic type system that is implemented for Java. Currently the implementation does not support "from" annotations; however, adding support for them appears straightforward.

5.6 Comparisons

Compared to approached based solely on ownership, our system is able to detect when an iterator is invalidated.

Compared to approaches based on program verification, our permissions system is not as powerful. It is based on logic that is similar to power to decidable logics. It remains to be shown that our permission type system (not given here) is decidable. By abstracting out just the access to mutable state, permissions represent a more high-level view of the program than any system that uses model fields (say) to represent version stamps.

Compared to approaches based on linear logic, our system is simpler because it uses information hiding (ownership). Both Bierhoff's and Krishnaswami's approaches have explicit mention of the collection state in the iterator state. Thus it appears that these systems could not support iterator patterns that use the iterator in places where the collection is unknown. Indeed while we require both aliasing annotations and effects annotations, the effect annotations on iterators are very simple. On the other hand, Bierhoff's system tracks type state too: a positive return from `hasNext()` ensures that `next()` can be called safely.

5.7 Other Applications

The "from" annotation is a general solution to the problem of how to grant temporary access to internals. Consider a buffered stream. Internally it has an unbuffered stream. Sometimes a client may wish to perform actions on the unbuffered stream and then resume using the buffered stream. One technique is for the client to be given access to the underlying stream and "hope" that the client will remember to flush the buffered stream before any unbuffered access. A less error-prone approach is to use an enforced "from" annotation:

```
class BufferedOutputStream
    implements OutputStream {
  ⋮
  writes(All)
  from(All) OutputStream getUnderlying() {
    flush();
    return underlying;
  }
}
```

Here the underlying stream encumbers the buffered output stream. If the client wishes to use the buffered stream again, it must give up access (permission) to the underlying stream. Thus we see that "from" is a general solution for a class of problems.

6. Conclusion

We have informally described the concept of permissions which combines an ownership-like system (nesting) with linear types, and is flow sensitive. Permissions can be used to express the design intent on our examples; it can be enforced that a collection cannot be modified while non-mutating iterators are active. The system is flexible enough to permit several interesting iterator usage patterns with minor annotation overhead. The "from" annotation can also be used more generally whenever a class wishes to grant temporary access to internal data structures.

References

[1] Jonathan Aldrich and Craig Chambers. Ownership domains: Separating aliasing policy from mechanism. In Martin Odersky, editor, *ECOOP'04 — Object-Oriented Programming, 18th European Conference*, Oslo, Norway, June 14–18, volume 3086 of *Lecture*

Notes in Computer Science, pages 1–25. Springer, Berlin, Heidelberg, New York, 2004.

[2] Kevin Bierhoff. Iterator specification with typestates. In *SAVCBS '06: Proceedings of the 2006 conference on Specification and verification of component-based systems*, Portland, Oregon, USA, pages 79–82. ACM Press, New York, 2006.

[3] Chandrasekhar Boyapati. *SafeJava: A Unified Type System for Safe Programming*. Ph.D., Massachusetts Institute of Technology, February 2004.

[4] Chandrasekhar Boyapati, Alexandru Salcianu, William Beebee, and Martin Rinard. Ownership types for safe region-based memory management in Real-Time Java. In *Proceedings of the ACM SIGPLAN '03 Conference on Programming Language Design and Implementation*, San Diego, California, June 8–11, *ACM SIGPLAN Notices*, 38:324–337, May 2003.

[5] John Boyland. Checking interference with fractional permissions. In R. Cousot, editor, *Static Analysis: 10th International Symposium*, San Diego, California, USA, June 11-13, volume 2694 of *Lecture Notes in Computer Science*, pages 55–72. Springer, Berlin, Heidelberg, New York, 2003.

[6] John Boyland and William Retert. Connecting effects and uniqueness with adoption. In *Conference Record of POPL 2005: the 32nd ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, Long Beach, California, USA, January 12-14, pages 283–295. ACM Press, New York, 2005.

[7] John Boyland, William Retert, and Yang Zhao. Comprehending annotations on object-oriented programs using fractional permissions. Submitted to OOPSLA '07. Manuscript available at http://www.cs.uwm.edu/faculty/boyland/papers/oo-permissions.pdf.

[8] Stephen Brookes. Variables as resource for shared-memory programs: Semantics and soundness. In *Twenty-second Conference on the Mathematical Foundations of Programming Semantics*, Genova, Italty, May 24–27, *Electronic Notes in Theoretical Computer Science*, 158:123–150, 2006.

[9] Dave Clarke and Sophia Drossopoulou. Ownership, encapsulation and the disjointness of type and effect. In *OOPSLA'02 Conference Proceedings—Object-Oriented Programming Systems, Languages and Applications*, Seattle, Washington, USA, November 4–8, *ACM SIGPLAN Notices*, 37(11):292–310, November 2002.

[10] David Clarke. *Object Ownership and Containment*. PhD thesis, University of New South Wales, Sydney, Australia, 2001.

[11] David G. Clarke, John M. Potter, and James Noble. Ownership types for flexible alias protection. In *OOPSLA'98 Conference Proceedings—Object-Oriented Programming Systems, Languages and Applications*, Vancouver, Canada, October 18–22, *ACM SIGPLAN Notices*, 33(10):48–64, October 1998.

[12] David R. Cok. Specifying Java iterators with JML and Esc/Java2. In *SAVCBS '06: Proceedings of the 2006 conference on Specification and verification of component-based systems*, Portland, Oregon, USA, pages 71–74. ACM Press, New York, 2006.

[13] W. Dietl and P. Müller. Universes: Lightweight ownership for JML. *Journal of Object Technology*, 4(8):5–32, October 2005.

[14] Manuel Fähndrich and Robert DeLine. Adoption and focus: Practical linear types for imperative programming. In *Proceedings of the ACM SIGPLAN '02 Conference on Programming Language Design and Implementation*, Berlin, Germany, June 17–19, *ACM SIGPLAN Notices*, 37:13–24, May 2002.

[15] Bart Jacobs, Frank Piessens, and Wolfram Schulte. VC generation for functional behavior and non-interference of iterators. In *SAVCBS '06: Proceedings of the 2006 conference on Specification and verification of component-based systems*, Portland, Oregon, USA, pages 67–70. ACM Press, New York, 2006.

[16] Neelakantan R. Krishnaswami. Reasoning about iterators with separation logic. In *SAVCBS '06: Proceedings of the 2006 conference on Specification and verification of component-based systems*, Portland, Oregon, USA, pages 83–86. ACM Press, New York, 2006.

[17] Peter Müller and Arsenii Rudich. Formalization of ownership transfer in universe types. Technical Report 556, ETH Zurich, 2007.

[18] James Noble. Iterators and encapsulation. In *TOOLS Europe 2000*, pages 431–442. IEEE Computer Society, Los Alamitos, California, 2000.

[19] Alex Potanin, James Noble, Dave Clarke, and Robert Biddle. Generic ownership for generic Java. In *OOPSLA'06 Conference Proceedings—Object-Oriented Programming Systems, Languages and Applications*, Portland, Oregon, USA, October 22–26, *ACM SIGPLAN Notices*, 41(10), October 2006.

[20] Matthew Smith. Toward an effects system for ownership domains. In *7th ECOOP Workshop on Formal Techniques for Java-like Programs*, Glasgow, UK, July 26. 2005.

[21] Bruce W. Weide. SAVCBS 2006 challenge: Specification of iterators. In *SAVCBS '06: Proceedings of the 2006 conference on Specification and verification of component-based systems*, Portland, Oregon, USA, pages 75–77. ACM Press, New York, 2006.