Ownership and Immutability in Generic Java (OIGJ)

Yoav Zibin Alex Potanin
Paley Li
Victoria University of Wellington
yoav|alex|lipale@ecs.vuw.ac.nz

Mahmood Ali
Massachusetts Institute of Technology
mahmood@mit.edu

Michael D. Ernst
University of Washington
mernst@cs.washington.edu

Abstract

The JAVA language has no support for the important notions of *ownership* (an object owns its representation to prevent unwanted aliasing or modifications) and *immutability* (the division into mutable, immutable, and readonly data and references). Programmers are prone to design errors such as representation exposure or violation of immutability contracts. This paper presents *Ownership Immutability Generic Java* (OIGJ), a backward-compatible purely-static language extension supporting ownership and immutability. We formally defined the OIGJ typing rules and proved them sound. We also implemented OIGJ and performed case studies of 33,000 lines of code.

OIGJ is the first type system to unify the two main variants of ownership: owner-as-*dominator*, which restricts aliasing, and owner-as-*modifier*, which restricts modifications. Thus, the programmer has the flexibility to choose whether the representation of an object cannot leak (the object *dominates* the representation) or whether it may be freely shared as readonly (but only the owning object can *modify* it).

OIGJ is easy for a programmer to use, and it is easy to implement (flow insensitive, using only 15 rules). Yet, OIGJ is more expressive than previous ownership languages, because it can type-check more good code. OIGJ can express the factory and visitor patterns, and OIGJ can type-check Sun's `java.util` collections (excluding the `clone` method) without refactoring and with only a small number of annotations. Previous work required major refactoring of existing code in order to fit its ownership restrictions. Forcing refactoring of well-designed code is undesirable because it costs programmer effort, results in worse design, and hinders adoption in the mainstream community.

Categories and Subject Descriptors CR-number [*subcategory*]: third-level

General Terms term1, term2

Keywords keyword1, keyword2

1. Introduction

This paper presents *Ownership Immutability Generic Java* (OIGJ), a simple and practical language extension that expresses both ownership and immutability information. OIGJ is purely static, without any run-time representation. This enables executing the resulting code on any JVM without runtime penalty. Our ideas, though demonstrated using Java, are applicable to any statically typed language with generics, such as C++, C#, Scala, and Eiffel.

Two standard ownership disciplines are owner-as-dominator and owner-as-modifier. The *owner-as-dominator* discipline [2, 11, 30] ensures that an object cannot leak beyond its owner: outside objects cannot access it. If an object owns its representation, then there are no aliases to its internal state. For example, a `LinkedList` should own all its `Entry` objects (but not its elements); entries should not be exposed to clients, and entries from different lists must not be mixed.

Restricting aliasing is not always desirable. Sharing of objects is common practice, e.g., collections with iterators, and the flyweight and model-view-controller patterns [16]. Instead, it suffices to enforce the weaker *owner-as-modifier* discipline [14, 21]: an object can be *mutated* only by its owner, but *readonly* references can be shared without restriction. This allows the owner to control state changes of its owned objects, for instance, in order to maintain invariants.

Previous work on ownership supported only one of these two disciplines. OIGJ permits an a-la-carte approach: different objects in a program can obey different ownership disciplines, owner-as-dominator or owner-as-modifier, depending on the need. Each object obeys just one ownership discipline and has exactly one owner. That owner is either the dominator or modifier of the object.

OIGJ is based on our previous work on ownership (OGJ [30]) and immutability (IGJ [35]). Although ownership and immutability may seem like two unrelated concepts, a design

involving both enhances the expressiveness of each individual concept.

On the one hand, adding immutability on top of ownership enables relaxing owner-as-dominator to owner-as-modifier. In our context, this removes limitations of OGJ.

On the other hand, the benefits of adding ownership on top of immutability have not been investigated before. One such benefit is easier creation of immutable cyclic data-structures by using ownership information. Constructing an immutable object must be done with care. The terms *raw* and *cooked* [7] describe the state of an immutable object: field assignment is allowed only when the object is *raw*, otherwise we say that the object is *cooked* and it cannot be further modified. An immutable object should not be visible to the outside world in its raw state because it would seem to be mutating. The challenge in building an immutable cyclic data-structures is that many objects must be raw simultaneously to create the cyclic structure. Previous work restricted cooking an object to the constructor, i.e., an object becomes cooked when its constructor finishes.

Our key observation is that *an object becomes cooked when its owner's constructor finishes*. More precisely, in OIGJ, a programmer can choose between cooking an object until its constructor finishes, or until its owner becomes cooked. Because the object is encapsulated within its owner, the outside world will not see this cooking phase. By adding ownership information, we can prolong the cooking time to make it easier to create complex data-structures.

Consider, e.g., building an immutable `LinkedList` from a given collection `c` (this code is similar to Sun's implementation):

```
Entry<E> succ = this.header;
for (E e : c) { ... succ.next = ...; succ.prev
= ...; }
```

An immutable list contains immutable entries, i.e., the fields `next` and `prev` cannot be changed after an entry is cooked. IGJ and previous work on immutability lack ownership information, and therefore cannot use such code to create an immutable list, because an entry would become cooked after its constructor finishes. In contrast, in OIGJ the above code type-checks if we specify that the list owns all its entries (the entries are the list's representation). The entries will become cooked when their owner's (the list's) constructor finishes, thus permitting the above assignment to `next` and `prev` during the list's construction. Therefore, there was no need to refactor the constructor of `LinkedList` for the benefit of OIGJ type-checking.

OIGJ can be expressed either using *generics syntax*, or using Java 7's *type annotations syntax* [15]. This paper presents OIGJ using *generics* to reduce the number of type rules; our OIGJ implementation uses *annotations* to reduce the programming overhead. By using default annotations (which is not possible when using generics), we were able to annotate Sun's `LinkedList` using only 3 ownership annotations (see Sec. 5). Because OIGJ is purely static, it does not support runtime casts that constrain immutability or ownership.

Contributions The main contributions of this paper are:

Dominators and modifiers The paper naturally unifies the two main ownership disciplines: owner-as-dominator and owner-as-modifier. In OIGJ, the owner generic parameter may be either `Dominator` or `Modifier`, denoting that the `this` object is the dominator or modifier, respectively.

Simplify ownership concepts OIGJ simplifies previous ownership concepts, such as scoped regions and existential owners [33, 34], by using the underlying *generic* mechanisms. Specifically, scoped regions are implemented using *generic methods*, and existential owners are implemented using *generic wildcards*. Moreover, OIGJ removes OGJ's limitation on owner-polymorphic methods, and allows generic wildcards as owners for stack variables.

No refactoring of existing code We have implemented OIGJ, annotated the collection classes (`java.util`, excluding the `clone` method), and verified that they are properly encapsulated by running the OIGJ type-checker. Previous approaches to ownership or immutability required major refactoring of this codebase. Refactoring of well-designed code costs programmer effort, results in worse design, and hinders adoption in the mainstream.

Flexibility As illustrated by our case study, OIGJ is more flexible and practical than previous type systems. For example, OIGJ can type-check the factory and visitor design patterns (see Sec. 3), but other ownership languages cannot [23]. Another reason that OIGJ can type-check more good code is that it uses ownership information to facilitate creating immutable objects.

Formalization An accompanying technical report [29] presents the main proofs for a core calculus of OIGJ.

Outline. Sec. 2 explains how owner-as-dominator and owner-as-modifier were unified. Sec. 3 presents OIGJ and how it enables the implementation of Factory and Visitor patterns. Sec. 4 discusses OIGJ formalization. Sec. 5 discusses the OIGJ implementation and the collections case study. Sec. 6 compares OIGJ to related work, and Sec. 7 concludes.

2. Combining Dominators and Modifiers

This section explains about two variants of ownership supported in OIGJ: owner-as-dominator and owner-as-modifier. It also formalize the runtime properties guaranteed by OIGJ.

To explain owner-as-modifier, we need to define terms related to immutability. Objects in OIGJ are either mutable or immutable. OIGJ static type system guarantees the runtime property that the fields of an immutable objects cannot be reassigned after the object is cooked. An object is cooked either when its constructor finishes or when its owner is

cooked. References in OIGJ are either mutable, immutable, or readonly. A readonly reference may point to mutable or immutable objects, and it cannot be used to mutate the object (although the object might still be mutated by other aliases).

Ownership is used to enforce proper encapsulation in terms of aliasing or modifications, i.e., it ensures that an object cannot be aliased or modified outside its owner. Fig. 1a presents an imaginary example describing school students that can enroll to different courses. A school object contains a list of students. A course object may point to students that are enrolled in the course. All objects in this example are mutable. We wish to use ownership in this example to restrict both aliasing and modification of objects: (i) only the school may point to (or modify) the list, and (ii) only the school may modify the students. Phrased differently, a course cannot point to the list nor modify the students, but it can point to (readonly) students. To summarize, the school owns both the list and the students, but it owns the list as a *dominator* (owner-as-dominator), and the students as a *modifier* (owner-as-modifier). Sec. 3 shows how OIGJ expresses such restrictions succinctly with the type: `LinkedList<Dominator,Mutable,Student<Modifier,Mutable>>`

Fig. 1b shows the *ownership tree*, while Fig. 1c shows the *dominator tree*. The ownership tree constrains object modifications (whether one object can *modify* another), and the dominator tree constrains heap structure (whether one object can *point to* another). Modifications must be done by the owner, no matter which ownership discipline is used.

The root of both trees is the `Stack` (the runtime execution stack), whose role is explained in Sec. 3. The ownership tree has two kinds of ownership edges, *modifier* and *dominator*, corresponding to whether an object follows the owner-as-modifier or owner-as-dominator discipline. For example, Fig. 1b shows that the school owns the students as a modifier and the list as a dominator.

The ownership tree is defined by the programmer, by assigning each object an owner (either as a dominator or modifier). To obtain the dominator tree, replace each modifier edge in the ownership tree by a dominator edge ending in `Stack`. If an object is dominated by `Stack` (e.g., the students in Fig. 1c), it may be freely shared.

If the ownership tree only has dominator edges, then the owner and the dominator are the same, and the whole program obeys the classical owner-as-dominator model. On the other hand, if it only has modifier edges, then the dominator is always `Stack`, and the whole program obeys the classical owner-as-modifier model.

In order to formalize the role of the dominator and ownership trees, we need to define some notation. Given an object o, its (direct) owner is $O(o)$ and its (direct) dominator is $D(o)$. For example, $O($`aList`$) =$ `aSchool`, $D($`aList`$) =$ `aSchool`, $O($`aStudent`$) =$ `aSchool`, and $D($`aStudent`$) =$ `Stack`. The ownership and dominator tree orders (both reflexive and transitive) are denoted by $\preceq_O$ and $\preceq_D$, respectively.

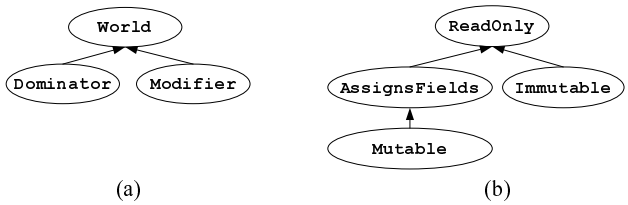

Figure 2. The type hierarchy of (a) ownership and (b) immutability parameters.

When $o_1 \preceq_O o_2$, then we say that o_1 is *inside* o_2, and o_2 is *outside* o_1. If neither object is inside another, then we say the objects are *incomparable*.

Given objects o_1 and o_2, we say that o_1 is a *transitive owner* of o_2 if $o_1 \preceq_O O(o_2)$ (similarly for *transitive dominator*). OIGJ guarantees the following runtime property: *An object can be mutated only by its transitive owners, and it can be pointed to only by its transitive dominators.* For example, the direct owner of `aNode` is `aList`, and the transitive owners of `aNode` are `aList` and all other nodes owned by `aList`. Therefore, `aNode` can point to (and mutate) `aList` and any other node.

Formally, we break down this property into heap and stack properties. Consider two objects o and o'.

Heap-property: o' can point to o iff o' is a transitive dominator of o.

Stack-property: o can be mutated iff there is a transitive *owner* of o on the call stack.

The stack-property permits mutation of objects as long as their transitive owner is on the call stack. For example, the owner of a collection may pass it to a sort method that temporarily alias and mutate it; this is allowed because the owner is on the call stack.

3. OIGJ Language

This section presents the OIGJ language extension that expresses both ownership and immutability information. We first describe the OIGJ syntax which is based on *conditional Java* (cJ) [18], where a programmer can guard methods with conditional type expressions. We then proceed with a `LinkedList` class example (Sec. 3.1), followed by the OIGJ typing rules (Sec. 3.2). We conclude by showing the factory (Sec. 3.3) and visitor (Sec. 3.4) patterns in OIGJ.

OIGJ introduces two new type parameters to each type, called the *owner parameter* and the *immutability parameter*. For simplicity of presentation, in the rest of this paper we assume that the special type parameters are at the beginning of the list of type parameters. We stress that generics in JAVA are erased during compilation to bytecode and do not exist at runtime, therefore OIGJ does not incur any runtime overhead.

In OIGJ, all classes are subtypes of the parameterized root type `Object<O,I>` that declares an owner and immutability parameter. All subclasses must invariantly preserve their

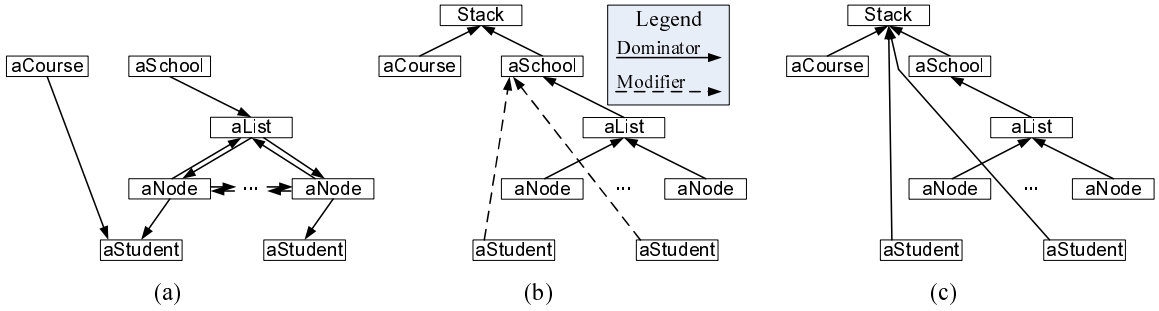

Figure 1. (a) Object graph for the school-course example, (b) its *ownership tree* with dominator and modifier edges, and (c) its *dominator tree*.

owner and immutability parameter. The owner and immutability parameters form two separate hierarchies, which are shown in Fig. 2. These parameters cannot be extended, and they have no subtype relation with any other types. The subtyping relation is denoted by $\preceq$, e.g., `Mutable` $\preceq$ `ReadOnly` .

Next we define the connection between the *compile-time* notion of the type parameters of Fig. 2 and the *run-time* notions of Sec. 2. With respect to immutability, this connection is easier to grasp because there is an obvious mapping from immutability parameters to immutability of objects: each reference has an immutability parameter (at compile-time) that hints on the immutability of the object (at run-time). Immutability parameter `Mutable` /`Immutable` imply the object is mutable/immutable, `ReadOnly` imply the object may be either mutable or immutable and thus cannot be mutated, whereas `AssignsFields` imply the object is still raw and thus can still be mutated but after it is cooked, it might become immutable.

With respect to ownership, this connection is harder to grasp because *owner parameters* (at compile-time) need to be mapped to *owners* (which are objects at run-time). Owner parameter `Dominator` imply the object is owned by `this` as a dominator (similarly for `Modifier`), whereas `World` imply the object is owned by the root of the ownership tree (so the entire world can access and modify it). Two references in the same class with the same owner parameter (at compile-time) will point to objects with the same owner (at run-time).

The following code shows an example of OIGJ syntax.

```
1: class Example<O extends World,I extends ReadOnly> {
2:   // An immutable reference to an immutable date.
     Date<O,Immutable> imD = new Date<O,Immutable>();
3:   // A mutable reference to a mutable date.
     Date<O,Mutable> mutD = new Date<O,Mutable>();
4:   // A this-mutable date: its mutability depends on the mutability of this
     Date<O,I> thisMutD = new Date<O,I>();
5:   // A readonly reference to any date.
     Date<O,ReadOnly> roD = ... ? imD : ... ? mutD : thisMutD;
6:   // Owner-as-dominator: only this can access (point to) this date; it cannot leak.
     Date<Dominator,Mutable> privateD = ...;
7:   // Owner-as-modifier: only this can mutate this date.
     Date<Modifier,Mutable> modifierD = ...;
8:   // Anyone can mutate this date.
     Date<World,Mutable> publicD = ...;
9:   // Anyone can access this date.
     Date<World,ReadOnly> sharedD = ... ? publicD : modifierD;

10:  int readonlyMethod() { ... }
11:  <I extends Mutable>? void mutatingMethod() { ... }
12: }
```

The owner and immutability parameters are declared on line 1; by convention we always denote them by `O` and `I` and they always extend `World` and `ReadOnly` . If the `extends` clause is missing from a class declaration, then we assume it extends `Object<O,I>` .

We use $O(\ldots)$ to denote the function that takes a type or a reference, and returns its owner parameter, e.g., $O($`privateD` $) =$ `Dominator` . Similarly, function $I(\ldots)$ returns the immutability parameter, e.g., $I($`privateD` $) =$ `Mutable` . We say that an object `o` is `this` *-owned* (i.e., owned by `this`) if its type contains `Dominator` or `Modifier` , e.g., `privateD` and `modifierD` are `this` -owned, but `mutD` is not. We say that an object `o` is `this` *-mutable* (its mutability depends on the mutability of `this`) if $I($`o`$) =$ `I`, e.g., `thisMutD` is `this` -mutable.

Lines 2–5 show different kinds of immutability in OIGJ: immutable, mutable, `this` -mutable, and readonly. The immutability of `thisMutD` (line 4) depends on the mutability of `this` , i.e., it is (im)mutable in an (im)mutable `Example` object. A readonly and an immutable reference may seem similar at first because neither can be used to mutate the referent. However, line 5 shows the difference between the two: a readonly reference may point to a mutable object. Phrased differently, a readonly reference may not mutate its referent, though the referent may be changed via an aliasing mutable reference.

Java's type arguments are no-variant, to avoid a type loophole [19], so line 5 is illegal in Java. Line 5 is legal in OIGJ, because OIGJ safely allows covariant changes in the immutability parameter. Therefore, neither OIGJ nor Java subsumes the other, i.e., a legal OIGJ program may be illegal in Java (and vice versa). However, because generics are erased during compilation, the resulting byte-code can be executed on any JVM.

Lines 6 and 7 show *owner-as-dominator* for `privateD` and *owner-as-modifier* for `modifierD`. Reference `privateD` is truly private, i.e., it cannot leak outside of `this`. The keyword `private` does not offer such strong protection, because a careless programmer might write a public method that exposes a private object (a.k.a. representation exposure). Phrased differently, the name-based protection used in JAVA hides the variable but not the object, as opposed to ownership that ensures proper encapsulation.

The key idea in ownership is that representation objects are nested and encapsulated inside the objects to which they belong. Because this nesting is transitive, this kind of ownership is also called *deep* ownership [13]. For instance, in a linked list, the entries are representation objects: the entries belong to the list and the list owns the entries.

As opposed to `privateD`, reference `modifierD` can leak outside of `this`, but only as a readonly reference. Phrased differently, only `this` can modify `modifierD`.

Lines 8 and 9 demonstrate using the owner parameter `World`, which means that the entire world can mutate or access the object. Line 9 also shows that OIGJ supports limited covariance in the owner parameter as well: `Date<Modifier,Mutable>` is a subtype of `Date<World,ReadOnly>`. However, such covariance is only allowed for `Modifier` and only when the subtype is readonly. For instance, the following two assignments are illegal `sharedD = privateD;` `sharedD = roD;` (The formal subtype definition is given in Sec. 3.2.)

Lines 10 and 11 show a readonly and a mutating method. The bound of `I` is declared on line 1 as `ReadOnly`, and it remains unchanged in `readonlyMethod`. However, the bound changes to `Mutable` in `mutatingMethod` by using the *conditional type expression* `<I extends Mutable>?`.

Conditional Java (cJ) [18] extends Java with conditional type expressions such as the above. Note that cJ changed Java's syntax by using the question mark in the above conditional `<...>?`. Our implementation of OIGJ (Sec. 5) uses type-annotations without changing Java's syntax.

Conditional type expressions such as `<T extends U>? X` intuitively means that `X` is included only if `T extends U`, and therefore inside the statement `X` one can assume that the bound of `T` is `U`. In our example, the conditional on line 11 has a dual affect: (i) inside the method the bound of `I` changes to `Mutable`, and (ii) it guards the method, i.e., this method can only be called on a mutable receiver. For example, (i) field `thisMutD` is mutable in `mutatingMethod`, and (ii) only a mutable `Example` object can call `mutatingMethod`. cJ also ensures that the condition of an overriding method is equivalent or weaker than the condition of the overriden method.

IGJ [35] used *method annotations* to denote the immutability of `this`. OIGJ is built on top of cJ to reduce the number of typing rules and handle inner classes more flexibly. OIGJ does not use the full power of cJ, i.e., it only uses conditionals with immutability parameters. Moreover, we modified cJ to treat conditional expressions over constructors in a special way described below in **Object creation rule** of Fig. 4.

3.1 `LinkedList` example

Fig. 3 shows an implementation of `LinkedList` in OIGJ that is similar in spirit to Sun's implementation. We explain this example in three stages: (i) we first explain the data-structure, i.e., the fields of a list and its entries (lines 1–5), (ii) then we discuss the `AssignsFields` constructor that enables creation of immutable lists (lines 6–19), and (iii) finally we dive into the complexities of inner classes and iterators (lines 21–46).

LinkedList data-structure A linked list has a header field (line 5) pointing to the first entry, where each entry has an element and pointers to the next and previous entries (line 2). We explain first the immutability and then the ownership of each field.

Recall that we implicitly assume that `O` extends `World` and that `I` extends `ReadOnly` on lines 1, 4, and 42.

An (im)mutable list contains (im)mutable entries, i.e., the entire data-structure is either mutable or immutable as a whole. Therefore, all the fields have the same immutability `I`, i.e., they are `this`-mutable. The underlying generic type system propagates the immutability information without the need for special type rules.

Next consider the ownership of the fields of list and entry. We use `Dominator` on line 5 to express that the reference `header` points to an `Entry` dominated by `this`, i.e., the entry is encapsulated and cannot be aliased outside of `this`. We use `O` on line 2 to express that the owner of `next` is the same as the owner of the entry, i.e., a linked-list owns *all* its entries. Note how the generics mechanism propagates the owner parameter, e.g., the type of `this.header.next.next` is `Entry<Dominator,I,E>`. Thus, the owner of all entries is the `this` object, i.e., the list.

Finally, note that the field `element` has no immutability nor owner paremeters, because they will be specified in the outside usage of the list, e.g.,
`LinkedList<Dominator,Mutable,Student<Modifier,ReadOnly>>`

Immutable object creation A constructor that is making an immutable object must be able to set the fields of the object. It is not acceptable to mark such constructors as mutable (`<I extends Mutable>?`), which would permit arbitrary side effects, possibly including making mutable aliases to `this`. OIGJ uses a fourth kind of reference immutabil-

```
1: class Entry<O,I__,E> {
2:   E element;  Entry<O,I___,E> next;  Entry<O,I___,E>
   prev;  ...
3: }
4: class LinkedList<O,I___,E> {
5:   Entry<Dominator,I___,E> header;
6:   <I extends AssignsFields>?      LinkedList() {
7:     this.header = new Entry<Dominator,I_____,E>();
8:     this.header.next = this.header; ...
9:   }
10:  <I extends AssignsFields>?      LinkedList(
11:                                   Collection<?,ReadOnly___,E>
   c) {
12:    this();
13:    this.addAll(c);
14:  }
15:  <I extends AssignsFields>?      void addAll(
16:                                   Collection<?,ReadOnly___,E>
   c) {
17:    Entry<Dominator,I___,E> succ = this.header;
18:    for (E e : c) { ...  succ.next = ...;  }
19:  }
20:  int size() {...}
21:  <ItrI extends ReadOnly>     Iterator<O,ItrI,I___,E>
   iterator() {
22:    return this.new ListItr<ItrI____>();
23:  }
24:  void remove(Entry<Dominator,Mutable_____,E> e) {
25:    e.prev.next = e.next;
26:    e.next.prev = e.prev;
27:  }
28:  class ListItr<ItrI___> implements
   Iterator<O,ItrI,I___,E> {
29:    Entry<Dominator,I___,E> current;
30:    <ItrI extends AssignsFields>?      ListItr() {
31:      this.current = LinkedList.this.header;
32:    }
33:    <ItrI extends Mutable>?     E next() {
34:      this.current = this.current.next;
35:      return this.current.element;
36:    }
37:    <I extends Mutable>?     void remove() {
38:      LinkedList.this.remove(this.current);
39:    }
40:  }
41: }
42: interface Iterator<O,ItrI,CollectionI_____,E> {
43:  boolean hasNext();
44:  <ItrI extends Mutable>?     E next();
45:  <CollectionI extends Mutable>?     void remove();
46: }
```

Figure 3. `LinkedList<O,I,E>` in OIGJ.

ity, `AssignsFields`, to permit constructors to perform limited side effects without permitting modification of immutable objects. Phrased differently, `AssignsFields` represents a *raw* object that can still be mutated, but after it is cooked, then the object might become immutable. An `AssignsFields` constructor can create both mutable and immutable objects.

Objects must not be captured in their raw state to prevent further mutation after the object is cooked. Therefore, a programmer can write the `AssignsFields` type only after the `extends` keyword, but *not* in any other way, such as `Date<?,AssignsFields>`. Note that, in an `AssignsFields` constructor, `this` can only escape as `ReadOnly`.

An object becomes *cooked* either (i) when its owner is *cooked* (if its immutability parameter is `I`, and its owner parameter is `Dominator` or `Modifier`), or (ii) when its constructor finishes (otherwise).

The constructors on lines 6 and 10 can create both mutable and immutable lists because they are guarded with `AssignsFields`. For example, the following creates an immutable list:

```
new LinkedList<?,Immutable___,Integer>( Arrays.asList(1,2,3)
)
```

This new list becomes cooked when its constructor finishes according to part (ii) in the definition of cooked.

In contrast, the entries of the list are of type `Entry<Dominator,I___,E>` (line 7), thus they will become cooked when the list is cooked according to part (i). Indeed, the entries are mutated after their constructor finished but before the list is cooked on lines 8 and 18. This shows the power of combining immutability and ownership: we are able to create immutable lists *only* by using the fact that the list owns its entries. If those entries were *not* owned by the list, then this mutation of entries might be visible to the outside world, thus breaking the guarantee that an immutable object never changes. By enforcing ownership, OIGJ ensures such illegal mutations cannot occur.

Note that we always access or assign to `header` via `this` (lines 7, 8, 17, and 31). OIGJ ensures that all access and assignment to a `this`-owned field (`header`) must be done via `this` (or else the resulting type is adapted by becoming readonly, see Sec. 3.2). In contrast, fields `next` or `prev` (which are not `this`-owned) do not have such restriction, as can be seen on lines 25–26.

Iterator implementation and inner classes An iterator has an underlying collection, and the immutability of the two might be different. For example, you can have a mutable iterator to a readonly collection (i.e., you can call `next()` but not `remove()`), or a readonly iterator to a mutable collection (i.e., you can call `remove()` but not `next()`). Consider the `Iterator<O,ItrI,CollectionI,E>` interface on lines 42–46. Note that `ItrI` is the iterator's immutability, whereas `CollectionI` is the underlying collection's immutability. Line 44 requires a mutable iterator in order to call `next()`, and line 45 requires a mutable collection to call `remove()`.

Although the immutability of the iterator and its collection might be different, we assume that their owner parameter `O` is the same. The reason is that the iterator directly accesses the (`this`-owned) representation of the collection, which would be illegal if their owner was different.

Inner class `ListItr` (lines 28–40) is the implementation of `Iterator` for list. It inherits the owner parameter `O` from `LinkedList`, but declares a new immutability parameter `ItrI`, and uses them on line 28 in `Iterator<O,ItrI,I,E>`.

In general, we found that an inner class should have distinct immutability parameter, but it should inherit the owner parameter of its outer class. This design decision is reinforced by the fact that the immutability typing rules do not use `this`, whereas the ownership typing rules use `this` when checking if field assignment or access are legal. The fact that the owner parameter of the inner and outer instances is the same, make it easy to type-check statements involving two distinct `this` objects, the iterator (`this`) and the list (`LinkedList.this`), such as on lines 31 and 38. Moreover, the semantics of immutability has no relationship with `this`: an immutable entry is immutable whether it appears in an outer, inner or another class. In contrast, the semantics of `Dominator` and `Modifier` are related to `this`, e.g., `Entry<Dominator,I,E>` means the entry is dominated by `this`, and the semantics should be the same whether this type appears in the outer or inner class. We achieve this by treating an object and its inner instances as a single node in the ownership tree.

Notice the difference in syntax between a generic method on line 21 and a conditional on a method (that ends in a question-mark) on line 15.

Finally, consider the creation of a new *inner* object on line 22 using `this.new ListItr<ItrI>()`. This expression is type-checked both as a method call (whose receiver is `this`) and as a constructor call.

3.2 OIGJ typing rules

Fig. 4 contains all the OIGJ typing rules. In what follows we provide more detailed discussion of each of the rules. A formal type system based on these rules is contained in the accompanying technical report [29].

Ownership nesting Recall that OIGJ defines both an ownership tree and a dominator tree (see Fig. 1). *Ownership nesting* (for both owner-as-dominator and owner-as-modifier) is defined over the *dominator tree* in order to prevent breaking the dominator property. Recall that when the owner parameter is `Modifier` or `World` then the object is dominated by the `Stack` (the root of the dominator tree). E.g., according to ownership nesting, the definition of `l1` is legal but that of `l2` is illegal:

```
List<World,Mutable,Date<Modifier>, Mutable>> l1;     // OK
List<World,Mutable,Date<Dominator>,Mutable>> l2;     // Illegal!
// l2 is illegal because dominated dates might leak, e.g., using o2
Object<World,Mutable> o2 = l2;
```

Field assignment Assigning to a field should respect both immutability and ownership constraints. Part (i) of the rule enforces immutability constraints: A field can be assigned only by a `Mutable` or `AssignsFields` reference.

Part (ii) enforces ownership constraints, i.e., `this`-owned fields can be assigned only via `this`. A subtle point to note is that the type of a field `f` might *contain* the owner parameter `Modifier`, but $O(\texttt{f}) \neq \texttt{Modifier}$, e.g., `List<O,I,Date<Modifier>,I>`. Assignment into such a field may only be done via `this` as well.

Field access Part (i) of the rule concerns with the transitivity of `AssignsFields`: it is transitive only for `this`-owned references. Otherwise, the object becomes readonly. For example, consider the assignment `this.header.next = ...` on line 8 of Fig. 3. The method is guarded by `AssignsFields`, thus `this` is `AssignsFields`. The type of the field access `this.header` is `Entry<Dominator,AssignsFields,E>`, i.e., it is `AssignsFields` and `this`-owned, and therefore the rule does not apply. If the object were not `this`-owned, then all immutability parameters would change to readonly.

Part (ii) of the rule handles access to `this`-owned fields. As long as access to such fields is done via `this`, then access is allowed. The challenge is what to do when a `this`-owned object escapes from `this`. Previous work prohibited such escape in owner-as-dominator (which is too restrictive), and changed the object to readonly in owner-as-modifier. Universes [14] supports only owner-as-modifier, and the process of changing an object to readonly is called *view-point adaptation* (because the view-point on the object changed from `this` to the outside world).

OIGJ follows Universes approach and always allow an object to escape as readonly. However, in owner-as-dominator, OIGJ ensures that the object cannot be stored on the heap, which will violate the owner-as-dominator property. This is done in the last part of the rule by changing all the owner parameters to wildcards.

For example, consider an imaginary class `Rectangle`, that is used to illustrate the process of view-point adaptation:

```
class Rectangle<O extends World,I extends ReadOnly>
{
  Point<Dominator,I> p;  // Adapted: Point<?,ReadOnly>
  boolean isEqualTo(Rectangle<?,ReadOnly> r) {
    return this.p.x==r.p.x && this.p.y==r.p.y; }
  Point<Modifier,I> modifierP;   // Adapted: Point<Modifier,Read
  Point<Dominator,Immutable> immutP;  // Adapted: Point<?,Immutal
  Point<O_,I> peerP;  // No adaptation needed because it is not this-owned
  List<O_,I,Point<Modifier,I>> list;
  // Adapted: List<O_,ReadOnly,Point<Modifier,ReadOnly>>
}
```

Previous work on owner-as-dominator often use the example of a rectangle that owns its two points. Field `p` is an example of such a `this`-owned point. Consider method `isEqualTo`, and compare the two field accesses: `this.p` and `r.p`. The second access leaks the privately owned point, and therefore was considered illegal by previous work, thus leading to inefficient solutions that copy the private point. OIGJ takes a different approach: the access `r.p` is allowed, but the point is leaked as readonly with a wildcard owner parameter, i.e., `Point<?_,ReadOnly>`, thus preventing anyone from mutating or storing the point on the heap.

In the above example, we wrote after each field its type after performing view-point adaptation, assuming it was accessed from a `Rectangle<O,I>`. Note that we adapt the owner parameter to a wildcard only for `Dominator`, not for `Modifier`. Also, there is no need to replace `Immutable` with `ReadOnly` for `immutP`. Finally, we note that view-point adaptation is applied recursively to the entire type, e.g., for field `list` we change both occurrences of `I` to `ReadOnly`. (The first occurrence of `I` was replaced as well to prevent mutating `list` and inserting a date with incorrect type.)

Method invocation Method invocation is handled in the same way as field access/assignment. For example, consider the following method: `R m(A a) { ... }`
Then, the method call `foo.m(e)` is handled as if there is an assignment to a field of type `A`, and the return value is typed as if there was an access to a field of type `R`.

Inner classes Nested classes that are *static* can be treated the same as normal classes. An *inner class* is a non-static nested class, e.g., iterators in `java.util` are implemented using inner classes. An inner class inherits the owner parameter of the outer class, i.e., the inner object is seen as an extension of the outer object. However, it has a distinct immutability parameter. Therefore, both `this` and `OuterClass.this` are treated identically by the type rules that involve ownership. (Note that the type rules for immutability never mention "`this`".)

Inheritance OIGJ prohibits defining *manifest classes*, which are classes without an ownership or immutability parameter. For instance, the following definition is illegal:

```
class MutableDate<O extends World> extends Date<O,Mutable>
{ }
```

Recall that view-point adaptation changes all immutability parameters to readonly in order to enforce owner-as-modifier. Defining `MutableDate` is illegal because it lacks an immutability parameter. More generally, the **Inheritance rule** prohibits any fixed immutability or ownership parameters in the supertype:

```
class MutableDateList<O,I,DateO> extends
                    List<O,I,Date<DateO,Mutable>> { }
// Illegal!
class DateList<O,I,DateO,DateI> extends
                    List<O,I,Date<DateO,DateI>> { }      //
Legal
```

Manifest classes are usually used to save annotation overhead. This overhead can be saved in OIGJ by using default annotations (Sec. 5).

Subtype relation JAVA is *no-variant* in generic arguments, i.e., it prohibits *covariant* (or contra-variant) changes. A `Vector<Integer>` is not a subtype of a `Vector<Object>`. If it were, then mutating the vector by inserting, e.g., a `String`, breaks type-safety.

OIGJ permits covariant changes for non-mutable references because the object cannot be mutated in a way that is not type-safe. The full subtype definition of OIGJ includes all of Java's subtyping rules, therefore OIGJ's subtype relation is a superset of Java's subtype relation. We only present a more relaxed same-class subtyping rule that allows covariant changes in other type parameters if mutation is disallowed, e.g., `List<O,ReadOnly,Integer>` is a subtype of `List<O,ReadOnly,Number>`. Note that covariance is allowed iff *all* immutability parameters of the supertype are `ReadOnly` or `Immutable`, e.g., `Iterator<O,ReadOnly,Mutable,Integer>` is *not* a subtype of `Iterator<O,ReadOnly,Mutable,Number>`, but it is a subtype of `Iterator<O,ReadOnly,ReadOnly,Number>`.

Requiring that *all* parameters are readonly can sometimes be too restrictive. For example, consider the class `DateList<O,I,DateO,DateI>` defined above. Then, `O` can change covariantly if `O` is readonly (irrelevant of `DateI`), however, `DateO` can only change covariantly if both `I` and `DateI` are readonly. The subtyping rule could be relaxed further at the cost of maintaining relationship between immutability parameters and other parameters. Bigger case studies are required in order to conclude if this is worthwhile. Finally, note that covariance in the owner parameter is limited by the requirement that: $\texttt{S}_i = \texttt{T}_i$ or $\texttt{S}_i =$ `Modifier`. E.g., `Date<Modifier,Mutable>` is a subtype of `Date<World,ReadOnly>`, but `Date<Dominator,Mutable>` is not a subtype of `Date<World,ReadOnly>` (because it could break owner-as-dominator).

No Variant A type parameter `X` in class `C` can be annotated with `@NoVariant` to prevent covariant changes, in which case we say that `X` is no-variant and write *NoVariant*(`X`, `C`). Otherwise we say that `X` is covariant and write *CoVariant*(`X`, `C`). Sometimes a type parameter must be no-variant, e.g., if it is used in a mutable field or if the erased signature differs (see below).

Erased signature When the erased signature of an overriding method differs from the overridden method, the normal `javac` compiler inserts a *bridge method* to cast the arguments to the correct type [8]. OIGJ requires that the *erased signature* of an overriding method remains the same if that method is either readonly or immutable. For example, in an implementation of `Comparable<O,ReadOnly,Integer>` the erased signature of `compareTo` differs from the one in the interface `Comparable<O,I,X>`. Therefore, this rule requires the type parameter `X` to be no-variant:

```
interface Comparable<O,I, @NoVariant X> { int compareTo(X o); }
```

Object creation One cannot create an immutable object from a mutable constructor, and vice versa. Recall that the immutability of a constructor (or any method in general) is defined to be the bound of the immutability parameter in that constructor, e.g., a mutable constructor will have the conditional `<I extends Mutable>?`. An `AssignsFields` constructor (i.e., a constructor guarded with `<I extends AssignsFields>?`) can create both mutable and immutable objects. Note that according to the rules of cJ such a call is illegal because the guard is not satisfied: `Immutable` is not a subtype of `AssignsFields`. OIGJ changed cJ to treat constructor calls using object creation rule.

CoVariant The immutability parameter must be allowed to change covariantly, or else a mutable reference could not be a receiver when calling a readonly method. Formally, *CoVariant*(`I`,`C`) must hold for any class `C`.

Generic Wildcards JAVA's generics can be bypassed by using reflection or raw types, e.g., `List`. Similarly, one can bypass OIGJ when using these features. OIGJ prohibits wildcards on the owner parameter of *fields*, e.g., `Date<?,ReadOnly> field`, because one can declare a static field of that type and store a `this`-owned date, thus breaking owner-as-dominator. Wildcards on method return type are also prohibited because they can be used to leak `this`-owned mutable fields. However, wildcards on *stack variables* (e.g., method parameters or local variables) are allowed.

Existential owners [9, 26, 34] are used when the exact owner of an object is unknown. One motivation for existential owners is the downcast performed in the `equals` method [34]. Without existential owners, this downcast requires a runtime check on the owner parameter.

OIGJ uses Java's existing generic wildcard syntax (`?`) to express existential owners. For example, consider a `DateList` class which is parameterized by its owner parameter (`O`) and the dates' owner parameter (`DO`):

```
class DateList<O extends World,DO extends World> {
  boolean equals(Object<?> o) {
    DateList<?,?> l = (DateList<?,?>) o;
    return listEquals(l);
  }
  <O2 extends World,DO2 extends World>
    boolean listEquals(DateList<O2,DO2> l) {...}
}
```

Method `listEquals` shows that it is possible to name the existential owner—the unknown list's owner parameter is `O2` and the unknown dates' owner parameter is `DO2`. Phrased differently, the two wildcards in `DateList<?,?>` are now named `DateList<O2,DO2>`.

`AssignsFields` *parameter* `AssignsFields` can only be used after the `extends` keyword. For example, it is prohibited to write directly `Date<O,AssignsFields>`. If it were possible, then such a date could leak from an `AssignsFields` constructor that is building an immutable object, and then that immutable object would have an alias that could mutate it.

Fresh owner A *fresh owner* is a method owner parameter that is not used in the method signature. In OIGJ, a fresh owner expresses *temporary ownership* within the method. Specifically, it allows a method to create stack-local objects with access to any object visible at the point of creation, but with a guarantee that stack-locals will not leak. Therefore, stack-local objects can be garbage-collected when the method returns. For example, consider a method that deserializes a `ByteStream` by creating a temporary `ObjectStream` that wraps it.

```
<O,TmpO> void deserialize(ByteStream<O> bs) {
  ObjectStream<TmpO,ByteStream<O>> os = ...        }
```

Note that `TmpO` is a fresh owner, whereas `O` is not. Because `TmpO` is strictly inside other owner parameters such as `O`, there cannot be any aliases from `bs` to `os`. When the method returns, `TmpO` will cease to exist, and the stack-local `os` can be garbage-collected.

Technically, *a fresh owner is strictly inside all other owners in scope*, to make sure it cannot exist after the method returns. Because a fresh owner is inside several other owners that might be incomparable in the ownership tree, the ownership structure is a DAG rather than a tree.

To type-check temporary ownership and DAG ownership structures, OIGJ adopts Wrigstad's *scoped confinement* [33] ownership model, in which the fresh owners are dominated by the current *stack-entry*. Briefly stated, each method invocation pushes a new stack-entry (the first stack-entry corresponds to the static `main` method), which is the root of a new ownership tree. Objects in this new tree may point to objects in previous trees, but not vice versa.

Static context `Dominator` and `Modifier` represent that an object is owned by `this`, and therefore OIGJ prohibits using them in a static context, such as static fields or methods. Static fields can use the owner parameter `World`, and static methods can also use generic method parameters that extends `World`. For example, the static method:

```
static <LO extends World> Collections.sort(List<LO,Mutable,E> l)
```

is parameterized by the list's owner `LO`.

3.3 Factory pattern

The *factory pattern* [16] is a creational design pattern for creating objects without specifying the exact class of object that will be created. The solution is to define an interface with a method for creating an object. Implementors can override the method to specify the derived type of object that will be created.

The challenge of the factory method with respect to ownership [23] is that the point of *creation* and *usage* are in different classes, and the created object must not be *captured* (stored in the fields of another object) between creation and

usage points. For example, consider a `this`-owned object that is *used* within some class, but *created* (and possibly *captured*, thus breaking ownership) outside the class. To solve this problem, previous work suggested sophisticated ownership transfer mechanisms [22] using `capture` and `release`.

OIGJ solves this problem without inventing a new mechanism. Specifically, *generic methods* can abstract over the owner (and immutability) parameter. The underlying generics mechanism finds the correct generic method arguments, and also ensures that the created object cannot be captured in the process.

We will show how to use the factory pattern in the context of *synchronized lists*. Consider this client code:

```
b = new LinkedList<T>();
l = Collections.synchronizedList(b);
```

The program may have concurrency problems if a programmer accidently uses the backing list `b` instead of `l`.[1]

Owner-as-dominator can be used to guarantee that the backing list `b` has no outside aliases. Phrased differently, you want to dominate a list `l`, which is backed-up by another list `b`. The challenge is that `b` should be owned by `l` (and not by you), thus guaranteeing that no one but `l` can access `b` (thus ensuring thread-safety).

Fig. 5 shows how ownership can ensure that the backing list cannot be accessed. This solution avoids refactoring of existing Java code by delegating calls to the synchornized list. Specifically, class `SyncList` (lines 1–9) owns both the list `l` (line 2) and the backing list `b` (line 5). The challenge is how to pass `b` from the outside, because the constructor on line 5 can only be called on `this` (line 6) due to the **Method invocation rule**. The key observation is that *if you want to own an object, you must create it*. This is achieved by using the factory method pattern on lines 3–7.

The `Factory` interface is defined on lines 15–17. The owner and immutability of the `Factory` is irrelevant because it only has a readonly method. However, the newly created list has a generic owner and immutability, which are unknown at the creation point (line 16). The generics mechanism fills in the correct generic arguments from the usage point (line 6) to the actual creation point (line 20). Note that the factory implementation cannot capture an alias to the newly created list on line 20. All this was achieved using *generic methods* on lines 16 and 19. To conclude, instead of using Sun's unsafe `synchronizedList`, one can use `SyncList` and be certain no one else can access the backing list.

[1] `Collections.unmodifiableList` has a similar problem when using it to create an immutable list, because the backing list might be mutated. In OIGJ, the list can declared as `Immutable` which statically enforce immutability without the runtime overhead of `unmodifiableList`. In contrast, in `synchronizedList`, the synchronization runtime overhead cannot be avoided.

3.4 Visitor pattern

The *visitor design pattern* [16] is a way of separating an algorithm from a node hierarchy upon which it operates. Instead of distributing the node processing code among all the node implementations, the algorithm is written in a single *visitor* class that has a visit method for every node in the hierarchy. This is desirable when the algorithm changes frequently or when new algorithms are frequently created. The standard implementation (that does not use reflection) defines a tiny `accept` method that is overridden in all the nodes, that calls the appropriate visit method for that node.

Nageli [23] discusses ownership in design patterns, and shows that previous work was not flexible enough to express the visitor pattern. A visitor is always mutable because it may accumulate information during the traversal of the nodes hierarchy. However, some visitors only need readonly access to the nodes, and some need to modify the nodes. In the former case, the owner of the visitor and nodes may be different, and in the latter case, it must be the same owner. The challenge is to use the same `visit` and `accept` methods, and to avoid duplicating the traversal code.

OIGJ can express the visitor pattern by relying (again) on generic methods. The key idea is to use owner-polymorphic methods: the owner of a mutable object `o`, can pass it to an *owner-polymorphic method*, who may mutate `o`, but may not alias `o` after the method returns.

Fig. 6 shows the visitor pattern in OIGJ. As mentioned before, the *owner* of the visitor and nodes may be different, and some visitors may or may not *modify* the nodes. Therefore, the visitor is parameterized on line 1 by the owner (`NodeO`) and immutability (`NodeI`) of the nodes. To make the code shorter, we omit the `extends` clause for generic parameters, e.g., we assume that `NodeO extends World`. The `visit` method on line 2 is mutable because it changes the visitor that accumulates information during the traversal. Different visitor implementations may have different immutability for the nodes, e.g., readonly on line 13 or mutable on line 21.

Finally, note how the type arguments `Modifier,ReadOnly` of the node on line 10 match the last two arguments of the visitor on line 12, and on line 18 the type arguments `Dominator,Mutable` match those on line 20. This shows that the same `accept` method (without duplicating the nodes hierarchy traversal code) can be used both for readonly and mutable hierarchies.

4. Formalization and Type Soundness

We formalize OIGJ and its guarantees in the context of a core calculus named Featherweight OIGJ, or FOIGJ. FOIGJ is based on Featherweight Generic Java (FGJ) [19], extended in a standard way [28] to support references (locations), heap (store), field assignment, and null. FOIGJ does not support wildcards, inner classes, or multiple immutability/owner parameters.

Due to space limitations, the details of the formalism, including syntax, auxiliary and lookup functions, well-formed types, subtyping, expression typing, and reduction rules, can be found in the accompanying technical report [29].

5. OIGJ Case Studies

This section describes our implementation of OIGJ: the language syntax (Sec. 5.1) and the type-checker implementation (Sec. 5.2). Sec. 5.3 presents our case study that involved annotating Sun's implementation of the `java.util` collections, and our conclusions about the design of the collection classes w.r.t. ownership and immutability.

5.1 Syntax: From Generics to Annotations

Whereas this paper uses generics to express ownership and immutability (e.g., `Date<O, I>`), our OIGJ implementation uses Java 7's type annotations (e.g., `@OI Date`), including receiver annotations instead of cJ's conditional type expressions.

Using annotations has the advantage of compatibility with existing compilers and other tools. Another advantage is the ability to use a default value, such as `@Mutable`. Furthermore, it is possible to customize these defaults per class. Defaults are not possible in generics, because one has to supply arguments for all generic parameters.

Using annotations has the disadvantage that some notions are no longer explicit in the syntax, such as transitivity, wildcards, and generic methods. To simulate generic methods in OIGJ such as:

```
<DI> Date<DI> getEarliest(List<ReadOnly,Date<DI>> l)
```

the annotation syntax uses `@I` with a string argument:

```
@I("DI") Date getEarliest(@ReadOnly List<@I("DI") Date> l)
```

Use of annotations also complicates the implementation (see below). For practical use, the compatibility benefits of using annotations outweigh their disadvantages.

OIGJ's annotations are the Cartesian product of owner parameters and immutability parameters. Due to a limitation of the Checker Framework [27], which can handle only a single type hierarchy, our implementation combines the ownership and immutability annotations — for example, `@DominatorI Date` instead of `@Dominator @I Date`. Our implementation does not yet support wildcards or classes with multiple owner or immutability parameters, such as `Visitor<O,I,NodeO,NodeI>` in Fig. 6.

A *class* can be annotated as `@Immutable` to indicate class immutability, i.e., all instances are immutable and no mutable methods exist.

5.2 OIGJ Implementation

The prototype OIGJ type-checker was implemented using the Checker Framework [27], which supports pluggable type systems using type annotations.

Because a pluggable type checker augments rather than replaces the type system of the underlying language, the Checker Framework permits only language extensions that are stricter than ordinary JAVA. A pluggable type system cannot relax JAVA's rules, such as the OIGJ subtyping rule. For example,

```
@Immutable List<@Immutable Date> a;
@ReadOnly List<@ReadOnly Date> b=a; // OK
@Immutable List<@Immutable Object> c=a; // Illegal!
```

The assignment `c=a` is illegal in Java and therefore in the Checker Framework, though it is legal in OIGJ itself. Phrased differently, in our implementation, the covariance is limited to annotations.

Most of the 1600 lines of code in the OIGJ type-checker is a reimplementation of the Generic Java type system. (The implementation of the ownership nesting rule and generic methods with immutability and ownership parameters is not yet complete.) By contrast, an implementation that directly modifies the `javac` compiler [30, 35] can reuse the generics type mechanisms for free.

5.3 `java.util` Collections Case Study

As a case study, we type-checked Sun's implementations of the `java.util` collections (77 classes, 33,246 lines of code). This required us to write 109 *ownership-related* annotations in 99 lines of code (the lines with `new` usually contain 2 annotations).

Sun's collections are not type-safe with respect to generics because generic arrays are not supported. However, the OIGJ implementation uses type annotations which can be placed on arrays as well, and therefore our annotated collections type-check without any errors with respect to ownership and immutability.

Class `LinkedList` in Fig. 3 is similar in essence to Sun's implementation. We annotated the constructors with `AssignsFields`, thus allowing creation of immutable instances. Since all instances of `Entry` are `this`-owned, using `@DominatorI` as the default annotation for `Entry` meant that only three *ownership-related*[2] annotations were needed in `LinkedList`:

```
@Default(DominatorI.class) static class Entry<E> {
  E element; @OI Entry<E> next; @OI Entry<E> previous; ... }
```

Similarly, in a `HashMap`, both the array and the entries are `this`-owned:

```
@DominatorI Entry[@DominatorI] table;
```

The case study supports these conclusions: (i) the collections classes are properly encapsulated (they own their representation as dominators), (ii) it is possible to create immutable instances (all constructors are `AssignsFields`), and (iii) methods `Map.get` and `clone` contain design mistakes (see below). We believe that if the collections were designed

[2] The other annotations are immutability-related, e.g., receiver annotations.

with ownership and immutability in mind, such mistakes could be avoided.

***Immutability of method* `get`** Let's start with a quick riddle: is there a `Map` implementation in `java.util` that might throw an exception when running the following *single-threaded* code?

```
for (Object key : map.keySet()) { map.get(key); }
```

The answer is that for a map created with

```
new LinkedHashMap(100, 1, /*accessOrder=*/ true)
```

that contains more than one element, the above code throws `ConcurrentModificationException` after printing one element.

Most programmers assume that `Map.get` is readonly, but there is no such guarantee in JAVA's specification. The documentation of `LinkedHashMap` states: "A special constructor is provided to create a linked hash map whose order of iteration is the order in which its entries were last accessed, from least-recently accessed to most-recently (*access-order*). Invoking the `put` or `get` method results in an access to the corresponding entry."

Because calling `get` modified the list, the above code threw `ConcurrentModificationException`. Phrased differently, method `LinkedHashMap.get` is mutable! Because an overriding method can only strengthen the specification of the overridden method, `HashMap.get` and `Map.get` must be mutable as well.

***Ownership and method* `clone`** Method `clone` violates owner-as-dominator because it leaks `this`-owned references by creating a shallow copy, i.e., only immediate fields are copied. Furthermore, Sun's implementation of `LinkedList` assigns to `result.header` which is a `this`-owned field. This violates **Field assignment rule**, which only permits assignment to `this.header`.

```
// Calling clone breaks owner-as-dominator because
// result.header is an alias to the this-owned field this.header
LinkedList result = (LinkedList) super.clone();
result.header = new Entry();    // Illegal in OIGJ!
```

Our technical report [29] describes a solution that, instead of initializing the cloned `result` from `this`, uses the idea of *inversion of control*. The solution has two parts. (1) The programmer writes a method `constructFrom` that initializes `this` from a parameter. (This is similar to a copy-constructor in C++, and indeed this method is given all the privileges of a constructor.) (2) The compiler automatically generates a `clone` method that first nullifies all the reference fields and then calls the user generated `constructFrom` method. This approach enforces the ownership and immutability properties.

6. Related Work

In this section we discuss the related work on ownership and immutability. We first highlight the relationship between OIGJ and our previous work on ownership (OGJ) and immutability (IGJ). We also survey some of the most relevant related language designs and show how OIGJ compares to them.

6.1 Relationship with OGJ and IGJ

OIGJ can be thought of as the "cartesian product" of OGJ and IGJ: OIGJ uses two type parameters to express ownership and immutability. However, the delicate intricacies between ownership and immutability required changes to both OGJ and IGJ, making OIGJ more expressive than a naive combination of both.

Ownership Generic Java (OGJ) [30] demonstrated how ownership and generic types can be unified as a language feature. OGJ featured a single owner parameter for every class that was treated in the same way as normal generic type parameters, simplifying both the language, the formalism, and implementation.

OGJ prohibits wildcards as owner parameters, e.g., `Point<?>`, because one can use such a type to store a `this`-owned point, thus breaking owner-as-dominator. OIGJ relaxes this rule and allows wildcards on stack variables, which enables writing the `equals` method (see **Generic Wildcards rule** in Sec. 3.2).

In OGJ, a method may have generic parameters that are owner parameters, e.g.,

```
class Foo<O extends World> {
  <O2 extends World __> void bar(Object<O2 __> o) {...}
```

However, OGJ required that the parametric owners are *outside* the owner of the class, e.g., $\texttt{O} \preceq_D \texttt{O2}$. This rule is very restrictive, however it prevents usages of owner polymorphism that create an ownership *directed acyclic graph* (DAG) instead of a tree. OIGJ removed this rule at the cost of complicating the ownership structure: DAG instead of a tree. Moreover, OIGJ can express temporary ownership within the method by using a fresh owner parameter (see **Fresh owners** in Sec. 3.2). Finally, OGJ only supports owner-as-dominator whereas OIGJ also supports owner-as-modifier.

Immutability Generic Java (IGJ) [35] showed how generic types can be used to provide support for readonly references and object immutability in a simple manner similar in spirit to OGJ.

The expressiveness of IGJ can be improved by using ownership information. In other words, certain restrictions in IGJ no longer apply in OIGJ for `this`-owned objects. For example, `AssignsFields` is *not* transitive in IGJ, e.g., the assignment to `next` in Fig. 3 on lines 8 and 18 is illegal in IGJ, thus limiting creation of immutable objects. In contrast, `AssignsFields` is transitive in OIGJ for `this`-owned fields (see **Field access rule**), and therefore there was no need to refactor the collections' code.

IGJ includes an `@Assignable` annotation on fields that permits field assignment even in immutable objects. OIGJ removed this annotation to guarantee that the fields of a cooked immutable object never change.

IGJ only permits a single immutability parameter which simplifies the subtyping rule. In contrast, types in OIGJ can have multiple immutability parameters, for example, `Iterator<O,ItrI,CollectionI,E>`. Because IGJ uses a single immutability parameter, the immutability of an iterator and its underlying collection must be the same. Thus, in IGJ, method `next()` must be readonly (or you couldn't iterate over a readonly list), and therefore we had to use an `@Assignable` annotation on `ListItr.current` (line 29 in Fig. 3). In contrast, in OIGJ we use cJ to guard `next()` with a mutable `ItrI` (line 44), and guard `remove()` with a mutable `CollectionI` (line 45). Furthermore, building OIGJ on top of cJ removed the following three typing rules from IGJ: (1) `this` rule, (2) Method overriding rule, and (3) Method invocation rule.

6.2 Relationship with Other Work

OIGJ uses conditional type expressions, borrowed from *cJ* [18], to guard method calls according to immutability parameters.

In what follows, we have room to survey only closely related papers from the rich literature on immutability and ownership. Fig. 7 compares OIGJ to some of the previous work described below.

Mutability and encapsulation were first combined by Flexible Alias Protection (FLAP) [25]. FLAP inspired a number of proposals including ownership types [12] and confined types [1]. Capabilities for Sharing [6] attempted to describe the fundamentals underlying various encapsulation and mutability approaches by separating "mechanism" (the semantics of sharing and exclusion) from "policy" (the guarantees provided by the resulting system). Capabilities gives a lower-level semantics that can be enforced at compile- or run-time. A reference can possess any combination of these 7 access rights: read, write, identity (permitting address comparisons), exclusive read, exclusive write, exclusive identity, and ownership (giving the capability to assert rights). Immutability, for example, is represented by the lack of the write right and possession of the exclusive write right. Finally, *Fractional Permissions* [7] can give semantics to various annotations such as unique, readonly, method effects, and an ownership variant called *owner-as-effector* in which one cannot read or write owned state without declaring the appropriate effect for the owner. The owner-as-dominator policy is structural, whereas owner-as-effector is semantic, thus allowing inlining method calls.

Ownership types [3, 4, 11] impose a structure on the references between objects in a program's memory. Two standard ownership disciplines are owner-as-dominator [26, 30] (also known as deep ownership [13]) and owner-as-modifier [14, 21]. Generic Universe Types (**GUT**) [14, 21] enforce owner-as-modifier by using three type annotations: `rep`, `peer`, and `readonly`. `rep` denotes representation objects, while `peer` denotes objects owned by the same owner. OIGJ subsumes this type system as follows: `rep Foo` corresponds to OIGJ's `Foo<Modifier,Mutable>`, `peer Foo` corresponds to `Foo<O,Mutable>`, and `any Foo` corresponds to `Foo<World,ReadOnly>`. **UTT** [22] is an extension of Universe Types that supports ownership transfer, which is useful for merging data-structures or complex object initialization.

MOJO [10] can express multiple ownership, i.e., objects can have more than one owner, resulting in an ownership DAG structure. OIGJ supports a single owner. Jo∃ [9] supports variant subtyping over the owner parameter by using existential types. OIGJ supports wildcards used as owners for stack variables, but those are less flexible than Jo∃. For example, Jo∃ can distinguish a list of students which may have different owners, from a list of student which share the same unknown owner.

Immutability and ownership. Similarly to OIGJ, Immutable Objects for a Java-like Language (**IOJ**) [17] associates with each type its mutability and owner. In contrast to OIGJ, IOJ does not have generics, nor readonly *references* (only readonly and immutable *objects*). Moreover, in IOJ, the constructor cannot leak a reference to `this`.

JOE$_3$ [26] combines ownership (as dominators, not modifiers), uniqueness, and immutability. It also supports owner-polymorphic methods, but not existential owners.

Readonly references are found in C++ (using the `const` keyword), JAC [20], modes [31], Javari [32], etc. Previous work on readonly references lack ownership information. Boyland [5] observes that readonly does not address observational exposure, i.e., modifications on one side of an abstraction boundary that are observable on the other side. Immutable objects address such exposure because their state cannot change.

List iterators pose a challenge to ownership because they require a direct pointer to the list's privately owned entries, thus breaking the ownership property. Both OIGJ and SafeJava [4] allow an inner instance to access the outer instance's privately owned objects. Clarke [11] suggested to use iterators only with stack variables, i.e., you cannot store an iterator in a field. It is also possible to redesign the code and implement iterators without violating ownership, e.g., by using internal iterators or magic-cookies [24].

7. Conclusion

OIGJ is a backward-compatible JAVA language extension that supports both ownership and immutability, while enhancing the expressiveness of each individual concept. By using JAVA's generic, OIGJ simplified previous type mechanisms, such as existential-owners, scoped regions, and owner-polymorphic methods. OIGJ is easy to understand and implement, using only 15 (flow-insensitive) typing rules. OIGJ supports both owner-as-dominator and owner-as-modifier, enabling a programmer to choose the most appropriate tool for the problem. OIGJ can type-check Sun's `java.util` collections (without the `clone` method), using a small number of annotations. Finally, various design pat-

	IOJ [17]	JOE_3 [26]	GUT [14], UTT [22]	OGJ [30]	IGJ [35]	OIGJ
Owner-as-dominator	+	+		+		+
Owner-as-modifier			+			+
Readonly references		+	+		+	+
Immutable objects	+	+			+	+
Factory pattern		+	+	+	+	+
Visitor pattern					+	+
Sun's `LinkedList`						+
Uniqueness		+				
Ownership transfer			+			

Figure 7. Features supported by various language designs.

terns, such as the factory and visitor patterns, can be expressed in OIGJ, making it ready for practical use.

Future work includes inferring ownership and immutability annotations, conducting a bigger case study including client and library code, and extending OIGJ with concepts such as *uniqueness* or *external-uniqueness* [13].

References

[1] Bokowski, B., Vitek, J.: Confined types. In: Proceedings of ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA). pp. 82–96. ACM Press (1999)

[2] Boyapati, C.: SafeJava: A Unified Type System for Safe Programming. Ph.D. thesis, MIT Dept. of EECS (Feb 2004)

[3] Boyapati, C., Lee, R., Rinard, M.: Ownership types for safe programming: Preventing data races and deadlocks. In: Proceedings of ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA). pp. 211–230. ACM Press (2002)

[4] Boyapati, C., Liskov, B., Shrira, L.: Ownership Types for Object Encapsulation. In: Proceedings of ACM Symposium on Principles of Programming Languages (POPL). pp. 213–223. ACM Press, New Orleans, LA, USA (Jan 2003), invited talk by Barbara Liskov.

[5] Boyland, J.: Why we should not add `readonly` to Java (yet). In: FTfJP (Jul 2005)

[6] Boyland, J., Noble, J., Retert, W.: Capabilities for sharing: A generalization of uniqueness and read-only. In: Proceedings of European Conference on Object-Oriented Programming (ECOOP). pp. 2–27. Springer-Verlag (Jun 2001)

[7] Boyland, J., Retert, W., Zhao, Y.: Comprehending annotations on object-oriented programs using fractional permissions. In: IWACO '09: International Workshop on Aliasing, Confinement and Ownership in Object-Oriented Programming. pp. 1–11. ACM (2009)

[8] Bracha, G., Odersky, M., Stoutamire, D., Wadler, P.: Making the future safe for the past: Adding genericity to the Java programming language. In: OOPSLA. pp. 183–200 (Oct 1998)

[9] Cameron, N., Drossopoulou, S.: Existential quantification for variant ownership. In: ESOP2009: European Symposium on Programming. pp. 128–142. Springer-Verlag (2009)

[10] Cameron, N., Drossopoulou, S., Noble, J., Smith, M.: Multiple ownership. In: Proceedings of ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA). pp. 441–460. ACM Press (2007)

[11] Clarke, D., Drossopoulou, S.: Ownership, Encapsulation, and the Disjointness of Type and Effect. In: Proceedings of ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA). pp. 292–310. ACM Press (Nov 2002)

[12] Clarke, D., Potter, J., Noble, J.: Ownership types for flexible alias protection. In: Proceedings of ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA). pp. 48–64. Vancouver, Canada (Oct 1998)

[13] Clarke, D., Wrigstad, T.: External uniqueness is unique enough. In: Proceedings of European Conference on Object-Oriented Programming (ECOOP). pp. 176–200 (Jul 2003)

[14] Dietl, W., Drossopoulou, S., Müller, P.: Generic Universe Types. In: Proceedings of European Conference on Object-Oriented Programming (ECOOP). pp. 28–53 (Aug 2007)

[15] Ernst, M.D.: Type annotations specification (JSR 308). `http://pag.csail.mit.edu/jsr308/` (Sep 12, 2008)

[16] Gamma, E., Helm, R., Johnson, R.E., Vlissides, J.: Design Patterns. Addison-Wesley (1994)

[17] Haack, C., Poll, E., Schäfer, J., Schubert, A.: Immutable objects for a Java-like language. In: ESOP. pp. 347–362 (2007)

[18] Huang, S.S., Zook, D., Smaragdakis, Y.: cJ: Enhancing Java with safe type conditions. In: AOSD. pp. 185–198. ACM Press (Mar 2007)

[19] Igarashi, A., Pierce, B.C., Wadler, P.: Featherweight Java: a minimal core calculus for Java and GJ. ACM TOPLAS 23(3), 396–450 (May 2001)

[20] Kniesel, G., Theisen, D.: JAC — access right based encapsulation for Java. Software: Practice and Experience 31(6), 555–576 (2001)

[21] Müller, P., Poetzsh-Heffter, A.: Universes: a Type System for Controlling Representation Exposure. In: Programming Languages and Fundamentals of Programming. pp. 131–140 (2001)

[22] Müller, P., Rudich, A.: Ownership transfer in universe types. In: Proceedings of ACM Conference on Object-Oriented

Programming, Systems, Languages, and Applications (OOPSLA). pp. 461–478. ACM Press (2007)

[23] Nägeli, S.: Ownership in Design Patterns. Master's thesis, ETH Zurich (2006)

[24] Noble, J.: Iterators and encapsulation. In: TOOLS Pacific (2000)

[25] Noble, J., Vitek, J., Potter, J.: Flexible Alias Protection. In: Proceedings of European Conference on Object-Oriented Programming (ECOOP). pp. 158–185 (Jul 1998)

[26] Östlund, J., Wrigstad, T., Clarke, D., Åkerblom, B.: Ownership, uniqueness and immutability. In: TOOLS Europe. pp. 178–197. Springer-Verlag (2008)

[27] Papi, M.M., Ali, M., Correa Jr., T.L., Perkins, J.H., Ernst, M.D.: Practical pluggable types for Java. In: ISSTA. pp. 201–212. Seattle, WA, USA (Jul 2008)

[28] Pierce, B.C.: Types and Programming Languages. MIT Press (2002)

[29] Potanin, A., Li, P., Zibin, Y., Ernst, M.D.: Featherweight Ownership and Immutability Generic Java (FOIGJ). Tech. Rep. 09-13, School of Engineering and Computer Science, VUW (2009), http://ecs.victoria.ac.nz/Main/TechnicalReportSeries

[30] Potanin, A., Noble, J., Clarke, D., Biddle, R.: Generic ownership for generic java. In: Proceedings of ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA). pp. 311–324. ACM Press, Portland, OR, USA (2006)

[31] Skoglund, M., Wrigstad, T.: A mode system for read-only references in Java. In: FTfJP (2001)

[32] Tschantz, M.S., Ernst, M.D.: Javari: Adding reference immutability to Java. In: OOPSLA. pp. 211–230 (Oct 2005)

[33] Wrigstad, T.: Ownership-Based Alias Management. Ph.D. thesis, Royal Institute of Technology, Sweden (May 2006)

[34] Wrigstad, T., Clarke, D.: Existential owners for ownership types. Journal of Object Technology (May 2007)

[35] Zibin, Y., Potanin, A., Artzi, S., Kiezun, A., Ernst, M.D.: Object and reference immutability using Java generics. In: FSE. pp. 75–84. ACM Press (2007)

Ownership nesting The owner parameter of a type must be lower or equal in the dominator tree compared to all other owner parameters inside that type.

Field assignment Field assignment `o.f=...` is legal iff (i) $I(\texttt{o}) \preceq \texttt{AssignsFields}$ and (ii) $\texttt{o} = \texttt{this}$ or `f` is not `this`-owned.

Field access Consider a field access `o.f`. If (i) $I(\texttt{o.f}) = \texttt{AssignsFields}$ and `o` is not `this`-owned, or (ii) `f` is `this`-owned and $\texttt{o} \neq \texttt{this}$, then the type of `o.f` is adapted by replacing all immutability parameters that are not `Immutable` with `ReadOnly`, and if $O(\texttt{f}) = \underline{\texttt{Dominator}}$ then we also replace all ownership parameters with the wildcard `?`.

Method invocation Typing method *parameters* is the same as field *assignment*, and typing the *return type* is the same as field *access*.

Inner classes An inner class inherits the owner parameter of the outer class. However, it has a distinct immutability parameter.

Inheritance Consider the definition of a class or interface `A` that extends or implements `B`. Then `B` cannot contain any fixed immutability or ownership parameters, i.e., it cannot contain `Dominator`, `Mutable`, etc, but it can contain variables such as `O`, `I`, etc.

Same-class subtype definition Let $\texttt{C<X}_1, \ldots, \texttt{X}_n\texttt{>}$ be a class. Type $\texttt{S} = \texttt{C<S}_1, \ldots, \texttt{S}_n\texttt{>}$ is *a subtype of* $\texttt{T} = \texttt{C<T}_1, \ldots, \texttt{T}_n\texttt{>}$, written as $\texttt{S} \preceq \texttt{T}$, iff $\texttt{C} = \texttt{S}$ or all immutability parameters $\texttt{T}_j$ are either `ReadOnly` or `Immutable`, and for $i = 1, \ldots, n$, $\texttt{S}_i \preceq \texttt{T}_i$ and $\mathit{CoVariant}(\texttt{X}_i, \texttt{C})$ and if $\texttt{T}_i$ is an owner parameter then $\texttt{S}_i = \texttt{T}_i$ or $\texttt{S}_i = \texttt{Modifier}$.

NoVariant A type parameter must be no-variant if it is used in a mutable superclass, a mutable field that is not `this`-owned, or in the position of another no-variant type parameter.

Erased signature If method $\texttt{m}'$ overrides method `m` and $\texttt{Immutable} \preceq I(\texttt{m})$, then the erased signatures of $\texttt{m}'$ and `m`, excluding no-variant type parameters, must be identical. (The *erased signature* of a method is obtained by replacing type parameters with their bounds.)

Object creation `new SomeClass<X,...>(...)` is *illegal* iff the immutability `Y` of the constructor satisfies:

$\texttt{Y} = \texttt{Mutable}$ and $\texttt{X} \neq \texttt{Mutable}$, or $\texttt{Y} = \texttt{Immutable}$ and $\texttt{X} \neq \texttt{Immutable}$.

CoVariant $\mathit{CoVariant}(\texttt{I}, \texttt{C})$ must hold for any class `C`.

Generic Wildcards OIGJ prohibits using generic wildcard (`?`) in the position of the immutability parameter. For the owner parameter, OIGJ prohibits using wildcards in fields or in method return type, but permits it for stack variables and method parameters.

`AssignsFields` parameter `AssignsFields` cannot be used in the position of a generic parameter. It can only be used after the `extends` keyword.

Fresh owners A *fresh owner* is a method owner parameter that is not used in the method signature. It is strictly inside all other owners in scope.

Static context `Dominator` and `Modifier` cannot be used in a static context, i.e., in static methods or fields.

Figure 4. OIGJ typing rules

```
1: class SyncList<O,I,E> implements List<O,I,E> {
2:   List<Dominator,I,E> l;
3:   <I extends AssignsFields>? SyncList(
4:                                     Factory<?,ReadOnly,E>
   f) {
5:     List<Dominator,I,E> b = f.create();
6:     l = Collections.synchronizedList(b);
7:   }
8:   ... // delegate methods to l
9: }
10: class Collections<O,I> {
11:   static <O2,I2,E>  List<O2,I2,E>
12:    synchronizedList(List<O2,I2,E> list)
13:      { ... }    // Sun's original implementation
14: }
15: interface Factory<O,I,E> {
16:   <O2,I2> List<O2,I2,E> create();
17: }
18: class LinkedListFactory<O,I,E> implements
   Factory<O,I,E> {
19:   <O2,I2>  List<O2,I2,E> create() {
20:     return new LinkedList<O2,I2,E>();
21:   }
22: }
```

Figure 5. Factory method pattern in OIGJ.

```
1: interface Visitor<O,I,NodeO,NodeI> {
2:  <I extends Mutable>? void
   visit(Node<NodeO,NodeI> n);
3: }
4: class Node<O,I> {
5:  void accept(Visitor<?,Mutable,O,I> v) {
6:    v.visit(this)
7:  }
8: }
9: // Visiting a readonly node hierarchy.
10: Node<Modifier,ReadOnly> readonlyNode = ...;
11: readonlyNode.accept(
12:  new Visitor<Modifier,Mutable,Modifier,ReadOnly>()
   {
13:   <I extends Mutable>? void
   visit(Node<Modifier,ReadOnly> n){
14:    ... // Can mutate the visitor, but not the nodes.
15:   }
16:  });
17: // Visiting a mutable node hierarchy.
18: Node<Dominator,Mutable> mutableNode = ...;
19: mutableNode.accept(
20:  new Visitor<Modifier,Mutable,Dominator,Mutable>()
   {
21:   <I extends Mutable>? void
   visit(Node<Dominator,Mutable> n){
22:    ... // Can mutate the visitor and the nodes.
23:   }
24:  });
```

Figure 6. Visitor pattern in OIGJ. The `Node` 's ownership and immutability are underlined.