Ownership and Immutability in Generic Java

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

Mahmood Ali
Massachusetts Institute of Technology
Cambridge, MA, USA
mali@csail.mit.edu

Michael D. Ernst
University of Washington
Seattle, WA, USA
mernst@cs.washington.edu

Abstract

The Java language lacks the important notions of *ownership* (an object owns its representation to prevent unwanted aliasing) 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 a core calculus for OIGJ, based on Featherweight Java, and proved it sound. We also implemented OIGJ and performed case studies on 33,000 lines of code.

Creation of immutable cyclic structures requires a "*cooking phase*" in which the structure is mutated but the outside world cannot observe this mutation. OIGJ uses *ownership* information to facilitate creation of *immutable* cyclic structures, by safely prolonging the cooking phase even after the constructor finishes.

OIGJ is easy for a programmer to use, and it is easy to implement (flow-insensitive, adding only 14 rules to those of Java). Yet, OIGJ is more expressive than previous ownership languages, in the sense that 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 (except for 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, degrades the design, and hinders adoption in the mainstream community.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features; D.1.5 [*Programming Techniques*]: Object-oriented Programming

General Terms Experimentation, Languages, Theory

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 run-time penalty. Our ideas, though demonstrated using Java, are applicable to any statically typed language with generics, such as C++, C#, Scala, and Eiffel.

OIGJ follows the *owner-as-dominator* discipline [1, 11, 33] where 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.

The keyword `private` does not offer such strong protection as ownership, 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 [12].

OIGJ is based on our previous type systems for ownership (OGJ [33]) and immutability (IGJ [40]). 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, immutability previously enhanced ownership by relaxing owner-as-dominator to owner-as-modifier [24]: it constrains modification instead of aliasing. 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. An object begins in the *raw* (not fully initialized) state and transitions to the *cooked* state [6] when initialization is complete. For an immutable object, field assignment is allowed only when the object is *raw*, i.e., the object cannot be modified after it is *cooked*. 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-structure 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 can be 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 building an immutable `LinkedList` (Sun's implementation is similar):

```
LinkedList(Collection<E> c)
{ this();
  Entry<E> succ = this.header, pred = succ.prev;
  for (E e : c)
  { Entry<E> entry = new Entry<E>(e,succ,pred);
    // It is illegal to change an entry after it is cooked.
    pred.next = entry;  pred = entry; }
  succ.prev = pred; }
```

An immutable list contains immutable entries, i.e., the fields `next` and `prev` cannot be changed after an entry is cooked. In IGJ and previous work on immutability, an object becomes cooked after its constructor finishes. Because `next` and `prev` are changed after that time, this code is illegal. In contrast, in OIGJ, this code type-checks if we specify that the list (the `this` object) 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 underlined assignments during the list's construction. Therefore, there was no need to refactor the constructor of `LinkedList` for the benefit of OIGJ type-checking.

Informally, OIGJ provides the following ownership and immutability guarantees. Let $\theta(o)$ denote the owner of o and let $\preceq_\theta$ denote the ownership tree, i.e., the transitive, reflexive closure of $o \preceq_\theta \theta(o)$. Phrased differently, $\theta(o)$ is the parent of o in the tree, and the top (biggest) node in the tree is the root `World`. We say that o_1 is *inside* o_2 iff $o_1 \preceq_\theta o_2$, and it is *strictly inside* if $o_1 \neq o_2$.

Ownership guarantee: An object o' can point to object o iff $o' \preceq_\theta \theta(o)$, i.e., o is owned by o' or by one of its owners. (This is the classical guarantee of *owner-as-dominator*.)

Immutability guarantee: An immutable object cannot be changed after it is cooked. (When exactly an object becomes cooked is formally defined in Sec. 3.)

Contributions The main contributions of this paper are:

Simplify ownership mechanisms Most previous works used new mechanisms that are orthogonal to generics. Our previous work (OGJ and IGJ) reuses Java's underlying generic mechanism. However, OGJ prohibited using generic wildcards and certain kinds of generic methods. OIGJ increases the expressiveness of OGJ without introducing new mechanisms, by implementing scoped regions [37] using *generic methods*, and existential owners [38] using *generic wildcards*.

No refactoring of existing code Java's collection classes (`java.util`) are properly encapsulated. We have implemented OIGJ, and verified the encapsulation by running the OIGJ type-checker without changing the source code (except for the `clone` method). Verifying Sun's `LinkedList` requires only 3 ownership annotations (see Sec. 4). Previous approaches to ownership or immutability required major refactoring of this codebase.

Flexibility As illustrated by our case study, OIGJ is more flexible and practical than previous type systems. For example, OIGJ can type-check the visitor design pattern (see Sec. 2), but other *ownership* languages cannot [26]. Another advantage of OIGJ is that it uses ownership information to facilitate creating immutable cyclic structures by prolonging their cooking phase.

Formalization We define a Featherweight OIGJ (FOIGJ) calculus to formalize the concepts of raw/cooked objects and wildcards as owner parameters. We prove FOIGJ is sound and show our ownership and immutability guarantees.

Outline. Sec. 2 presents the OIGJ language. Sec. 3 defines the OIGJ formalization. Sec. 4 discusses the OIGJ implementation and the collections case study. Sec. 5 compares OIGJ to related work, and Sec. 6 concludes.

2. OIGJ Language

This section presents the OIGJ language extension that expresses both ownership and immutability information. We first describe the OIGJ syntax (Sec. 2.1). We then proceed with a `LinkedList` class example (Sec. 2.2), followed by the OIGJ typing rules (Sec. 2.3). We conclude with the factory (Sec. 2.4) and visitor (Sec. 2.5) design patterns in OIGJ.

2.1 OIGJ syntax

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 run time, therefore OIGJ does not incur any run-time overhead (nor does it support run-time casts).

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

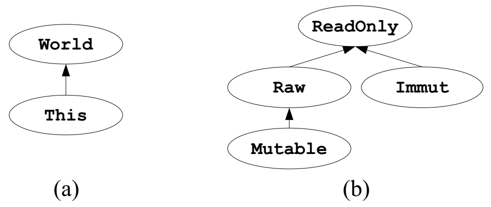

Figure 1. The type hierarchy of (a) ownership and (b) immutability parameters. `World` means the entire world can access the object, whereas `This` means that `this` owns the object and no one else can access it. The meaning of `Mutable`/`Immut` is obvious. A `ReadOnly` reference points to a mutable or immutable object, and therefore cannot be used to mutate the object. `Raw` represents an object under construction whose fields can be assigned.

The owner and immutability parameters form two separate hierarchies, which are shown in Fig. 1. These parameters cannot be extended, and they have no subtype relation with any other types. The subtyping relation is denoted by $\leq$, e.g., `Mutable` $\leq$ `ReadOnly`. Subtyping is invariant in the owner parameter and covariant in the immutability parameter. (See also paragraph ***Subtype relation*** in Sec. 2.3.)

Note that the *owner parameter* `O` is a *type*, whereas the *owner* of an object is an *object*. For example, if the owner parameter is `This`, then the owner is the object `this`. Therefore, the owner parameter (which is a type) at compile time corresponds to an owner (which is an object) at run time. (See also paragraph ***Owner vs. Owner-parameter*** below.)

OIGJ syntax borrows from *conditional Java* (cJ) [20], where a programmer can write method *guards*. A guard of the form `<X extends Y>? METHOD_DECLARATION` has a dual meaning: (i) the method is applicable only if the type argument that substitutes `X` extends `Y`, and (ii) the bound of `X` inside `METHOD_DECLARATION` changes to `Y`. The guards are used to express the immutability of `this`: a method receiver or a constructor result. For example, a method guarded with `<I extends Mutable>?` means that (i) the method is applicable only if the receiver is mutable and therefore (ii) `this` can be mutated inside the method.

Class definition example Fig. 2 shows an example of OIGJ syntax. A class definition declares the owner and immutability parameters (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>`.

Immutability example Lines 2–4 show different kinds of immutability in OIGJ: immutable, mutable, and readonly. A readonly and an immutable reference may seem similar because neither can be used to mutate the referent. However, line 4 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.

```
 1:class Foo<O extends World,I extends ReadOnly> {
 2:  // An immutable reference to an immutable date.
     Date<O,Immut> imD = new Date<O,Immut>();
 3:  // A mutable reference to a mutable date.
     Date<O,Mutable> mutD = new Date<O,Mutable>();
 4:  // A readonly reference to any date. Both roD and imD cannot
     mutate their referent, however the referent of roD might be
     mutated by an alias, whereas the referent of imD is immutable.
     Date<O,ReadOnly> roD = ... ? imD : mutD;
 5:  // A date with the same owner and immutability as this.
     Date<O,I> sameD;
 6:  // A date owned by this; it cannot leak.
     Date<This,I> ownedD;
 7:  // Anyone can access this date.
     Date<World,I> publicD;
 8:  // Can be called on any receiver; cannot mutate this. The
     method guard "<...>?" is part of cJ's syntax [20].
     <I extends ReadOnly>? int readonlyMethod() {...}
 9:  // Can be called only on mutable receivers; can mutate this.
     <I extends Mutable>? void mutatingMethod() {...}
10:  // Constructor that can create (im)mutable objects.
     <I extends Raw>? Foo(Date<O,I> d) {
11:    this.sameD = d;
12:    this.ownedD = new Date<This,I>();
13:    // Illegal, because sameD came from the outside.
       // this.sameD.setTime(...);
14:    // OK, because Raw is transitive for owned fields.
       this.ownedD.setTime(...);
15:}}
```

Figure 2. An example of OIGJ syntax.

Java's type arguments are invariant (neither covariant nor contravariant), to avoid a type loophole [21], so line 4 is illegal in Java. Line 4 is legal in OIGJ, because OIGJ safely allows covariant changes in the immutability parameter (but not in the owner parameter). OIGJ *restricts* Java by having additional typing rules, while at the same time OIGJ also *relaxes* Java's subtyping relation. 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 bytecode can be executed on any JVM.

The immutability of `sameD` (line 5) depends on the immutability of `this`, i.e., `sameD` is (im)mutable in an (im)mutable `Foo` object. Similarly, the owner of `sameD` is the same as the owner of `this`.

Ownership example Lines 5–7 show three different owner parameters: `O`, `This`, and `World`. The owner parameter is invariant, i.e., the subtype relation preserves the owner parameter. For instance, the types on lines 5–7 have no subtype relation with each other because they have different owner parameters.

Reference `ownedD` cannot leak outside of `this`, whereas references `sameD` and `publicD` can potentially be accessed by anyone with access to `this`. Although `sameD` and `publicD` can be accessed by the same objects, they cannot be stored in the same places: `publicD` can be stored anywhere on the heap (even in a static public variable) whereas `sameD` can only be stored inside its owner.

We use $O(\ldots)$ to denote the function that takes a type or a reference, and returns its owner parameter; e.g., $O(\texttt{ownedD}) = \texttt{This}$. Similarly, function $I(\ldots)$ returns the immutability parameter; e.g., $I(\texttt{ownedD}) = \texttt{I}$. We say that an object `o` is `this`-*owned* (i.e., owned by `this`) if $O(\texttt{o}) = \texttt{This}$; e.g., `ownedD` is `this`-owned, but `sameD` is not. OIGJ prevents leaking `this`-owned objects by requiring that `this`-owned fields (and methods with `this`-owned arguments or return-type) can only be used via `this`. For example, `this.ownedD` is legal, but `foo.ownedD` is illegal.

Owner vs. owner-parameter Now we explain the connection between the *owner parameter* $O(o)$, which is a generic type parameter at *compile time*, and the *owner* $\theta(o)$, which is an object at *run time*. `This` is an owner parameter that represents an owner that is the current `this` object, and `World` represents the root of the ownership tree (we treat `World` both as a type parameter and as an object that is the root of the ownership tree). Formally, if $O(o) = \texttt{This}$ then $\theta(o) = \texttt{this}$, if $O(o) = \texttt{O}$ then $\theta(o) = \theta(\texttt{this})$, and if $O(o) = \texttt{World}$ then $\theta(o) = \texttt{World}$. 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), i.e., $O(o_1) = O(o_2)$ implies $\theta(o_1) = \theta(o_2)$.

Finally, recall the **Ownership guarantee**: o' can point to o iff $o' \preceq_\theta \theta(o)$. By definition of $\preceq_\theta$, we have that for all o: (i) $o \preceq_\theta o$, (ii) $o \preceq_\theta \theta(o)$, and (iii) $o \preceq_\theta \texttt{World}$. By part (iii), if $\theta(o) = \texttt{World}$ then anyone can point to o. On lines 5–7, we see that `this` can point to `ownedD`, `sameD`, `publicD`, whose owner parameters are `This`, `O`, `World`, and whose owners are `this`, $\theta(\texttt{this})$, `World`. This conforms with the ownership guarantee according to parts (i), (ii), and (iii), respectively. More complicated pointing patterns can occur by using multiple owner parameters, e.g., an entry in a list can point to an element owned by the list's owner, such as in `List<`$\underline{\texttt{This}}$`,I,Date<`$\underline{\texttt{O}}$`,I>>`.

There is a similar connection between the immutability type parameter (at compile time) and the object's immutability (at run time). Immutability parameter `Mutable` or `Immut` implies the object is mutable or immutable (respectively), `ReadOnly` implies the referenced object may be either mutable or immutable and thus the object cannot be mutated through the read-only reference. `Raw` implies the object is still raw and thus can still be mutated, but it might become immutable after it is cooked.

Method guard example Lines 8 and 9 show a readonly and a mutating method. These methods are *guarded* with `<...>?`. Conditional Java (cJ) [20] extends Java with such guards (a.k.a. conditional type expressions). Note that cJ changed Java's syntax by using the question mark in the guard `<...>`$\underline{\texttt{?}}$. The exposition in this paper uses cJ for convenience. However, our implementation of OIGJ (Sec. 4) uses type annotations [15] without changing Java's syntax, for conciseness and compatibility with existing tools and code bases.

A guard such as `<T extends U>? METHOD_DECLARATION` has a dual purpose: (i) the method is included only if `T extends U`, and (ii) the bound of `T` is `U` inside the method. In our example, the guard on line 9 means that (i) this method can only be called on a `Mutable` receiver, and (ii) inside the method the bound of `I` changes to `Mutable`. For instance, (i) only a mutable `Foo` object can be a receiver of `mutatingMethod`, and (ii) field `sameD` is mutable in `mutatingMethod`. cJ also ensures that the condition of an overriding method is equivalent or weaker than the condition of the overridden method.

IGJ [40] used *declaration annotations* to denote the immutability of `this`. In this paper, OIGJ uses cJ to reduce the number of typing rules and handle inner classes more flexibly.[1] OIGJ does not use the full power of cJ: it only uses guards with immutability parameters. Moreover, we modified cJ to treat guards over constructors in a special way described in the **Object creation rule** of Fig. 4.

To summarize, on lines 8–10 we see three guards that change the bound of `I` to `ReadOnly`, `Mutable`, and `Raw`, respectively. Because the bound of `I` is already declared on line 1 as `ReadOnly`, the guard on line 8 can be removed.

Constructor example The constructor on line 10 is guarded with `Raw`, and therefore can create both mutable and immutable objects, because all objects start their life cycle as raw. This constructor illustrates the interplay between *ownership* and *immutability*, which makes OIGJ more expressive than previous work on immutability. OIGJ uses ownership information to prolong the *cooking phase* for owned objects: the cooking phase of `this`-owned fields (`ownedD`) is longer than that of non-owned fields (`sameD`). This property is critical to type-check the collection classes, as Sec. 2.2 will show.

Consider the following code:

```
class Bar<O extends World,I extends ReadOnly>
  { Date<O,Immut> d = new Date<O,Immut>();
    Foo<O,Immut> foo = new Foo<O,Immut>(d); }
```

Recall our **Immutability guarantee**: an immutable object cannot be changed after it is *cooked*. A `This`-owned object is cooked when its owner is cooked (e.g., `foo.ownedD`). Any other object is cooked when its constructor finishes (e.g., `d` and `foo`). The intuition is that `ownedD` cannot leak and so the outside world cannot observe this longer cooking phase, whereas `d` is visible to the world after its constructor finishes and must not be mutated further. The constructor on lines 10–15 shows this difference between the assignments to `sameD` (line 11) and to `ownedD` (line 12): `sameD` can come from the outside world, whereas `ownedD` must be created inside `this`.

[1] Our implementation uses *type annotations* to denote immutability of `this`. A type annotation `@Mutable` on the receiver is similar to a cJ `<I extends Mutable>?` construct, but it separates the distinct roles of the receiver and the result in inner class constructors.

Thus, `sameD` cannot be further mutated (line 13) whereas `ownedD` can be mutated (line 14) until its owner is cooked.

An object in a raw method, whose immutability parameter is `I`, is still considered raw (thus the modified body can still assign to its fields or call other raw methods) iff the object is `this` or `this`-owned. Informally, we say that `Raw` is *transitive* only for `this` or `this`-owned objects. For example, the receiver of the method call `sameD.setTime(...)` is not `this` nor `this`-owned, and therefore the call on line 13 is illegal; however, the receiver of `ownedD.setTime(...)` is `this`-owned, and therefore the call on line 14 is legal.

2.2 `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 1–6), (ii) then we discuss the `Raw` constructors that enable creation of immutable lists (lines 7–24), and (iii) finally we dive into the complexities of inner classes and iterators (lines 27–53).

LinkedList data-structure A linked list has a header field (line 6) pointing to the first entry. Each entry has an `element` and pointers to the `next` and `prev` entries (line 3). 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, 5, 35 and 49.

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

Next consider the ownership of the fields of `LinkedList` and `Entry`. `This` on line 6 expresses that the reference `header` points to an `Entry` owned by `this`, i.e., the entry is encapsulated and cannot be aliased outside of `this`. `O` on line 3 expresses 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<This,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 parameters, because they will be specified by the client that instantiates the list type, e.g.,
`LinkedList<This,Mutable,Date<World,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`, which would permit arbitrary side effects, possibly including making mutable aliases to `this`. OIGJ uses a fourth kind of immutability, `Raw`, to permit constructors to perform limited side effects without permitting modification of immutable objects. `Raw` represents a partially-initialized *raw* object that can still be arbitrarily mutated, but after it is cooked (fully initialized), then the object might become immutable. The

```
 1: class Entry<O,I,E> {
 2:   E element;
 3:   Entry<O,I,E> next, prev;
 4: }
 5: class LinkedList<O,I,E> {
 6:   Entry<This,I,E> header;
 7:   <I extends Raw>? LinkedList() {
 8:     this.header = new Entry<This,I,E>();
 9:     header.next = header.prev = header;
10:   }
11:   <I extends Raw>? LinkedList(
12:                 Collection<?,ReadOnly,E> c) {
13:     this();  this.addAll(c);
14:   }
15:   <I extends Raw>? void addAll(
16:                 Collection<?,ReadOnly,E> c) {
17:     Entry<This,I,E> succ = this.header,
18:                     pred = succ.prev;
19:     for (E e : c) {
20:       Entry<This,I,E> en=new Entry<This,I,E>();
21:       en.element=e; en.next=succ; en.prev=pred;
22:       pred.next = en;  pred = en;  }
23:     succ.prev = pred;
24:   }
25:   int size() {...}
26:   // iterator is a generic method; this is not a cJ guard:
27:   <ItrI extends ReadOnly> Iterator<O,ItrI,I,E>
28:                 iterator() {
29:     return this.new ListItr<ItrI>();
30:   }
31:   void remove(Entry<This,Mutable,E> e) {
32:     e.prev.next = e.next;
33:     e.next.prev = e.prev;
34:   }
35:   class ListItr<ItrI> implements
36:         Iterator<O,ItrI,I,E> {
37:     Entry<This,I,E> current;
38:     <ItrI extends Raw>? ListItr() {
39:       this.current = LinkedList.this.header;
40:     }
41:     <ItrI extends Mutable>? E next() {
42:       this.current = this.current.next;
43:       return this.current.element;
44:     }
45:     <I extends Mutable>? void remove() {
46:       LinkedList.this.remove(this.current);
47:     }
48: } }
49: interface Iterator<O,ItrI,CollectionI,E> {
50:   boolean hasNext();
51:   <ItrI extends Mutable>? E next();
52:   <CollectionI extends Mutable>? void remove();
53: }
```

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

constructors on lines 7 and 11 are guarded with `Raw`, and therefore can create both mutable and immutable lists.

Objects must not be captured in their raw state to prevent further mutation after the object is cooked. If a programmer could declare a field, such as `Date<O,Raw>`, then a raw date could be stored there, and later it could be used to mutate a cooked immutable date. Therefore, a programmer can write the `Raw` type only after the `extends` keyword, but *not* in any other way. As a consequence, in a `Raw` constructor, `this` can only escape as `ReadOnly`.

Recall that an object becomes *cooked* either when its constructor finishes or when its owner is *cooked*. The entries of the list (line 6) are `this`-owned. Indeed, the entries are mutated after their constructor finished, but before the list is cooked, on lines 9, 22, and 23. 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 that such illegal mutations cannot occur.

OIGJ requires that all access and assignment to a `this`-owned field must be done via `this`. For example, see `header`, on lines 8, 9, 17, and 39. In contrast, fields `next` and `prev` (which are not `this`-owned) do not have such a restriction, as can be seen on lines 32–33.

Iterator implementation and inner classes An *iterator* has an underlying *collection*, and the immutability of these two objects might be different. For example, you can have

- a mutable iterator over a mutable collection (the iterator supports both `remove()` and `next()`),
- a mutable iterator over a readonly/immutable collection (the iterator supports `next()` but not `remove()`), or
- a readonly iterator over a mutable collection (the iterator supports `remove()` but not `next()`, which can be useful if you want to pass an iterator to a method that may not advance the iterator but may remove the current element).

Consider the `Iterator<O,ItrI,CollectionI,E>` interface defined on lines 49–53, and used on lines 27 and 36. `ItrI` is the iterator's immutability, whereas `CollectionI` is intended to be the underlying collection's immutability (see on line 36 how the collection's immutability `I` is used in the place of `CollectionI`). Line 51 requires a mutable `ItrI` to call `next()`, and line 52 requires a mutable `CollectionI` to call `remove()`.

Inner class `ListItr` (lines 35–48) is the implementation of `Iterator` for list. Its full name is `LinkedList<O,I,E>.ListItr<ItrI>`, and on line 35 it extends `Iterator<O,ItrI,I,E>`. It reuses the owner parameter $\underline{\texttt{O}}$ from `LinkedList`, but declares a new immutability parameter $\underline{\texttt{ItrI}}$. An inner class, such as `ListItr<ItrI>`, only declares an immutability parameter because it inherits the owner parameter from its outer class. `ListItr` and `LinkedList` have the same owner `O`, but different immutability parameters (`ItrI` for `ListItr`, and `I` for `LinkedList`). `ListItr` must inherit `LinkedList`'s owner because it directly accesses the (`this`-owned) representation of `LinkedList` (line 39), which would be illegal if their owner was different. For example, consider the types of `this` and `LinkedList.this` on line 39:

```
Iterator<O,ItrI,...> thisIterator = this;
LinkedList<O,I,...> thisList = LinkedList.this;
```

Because line 38 sets the bound of `ItrI` to be `Raw`, `this` can be mutated. By contrast, the bound of `I` is `ReadOnly`, so `LinkedList.this` cannot.

An inner class must have a distinct immutability parameter, but it must reuse the owner parameter of its outer class. We could have several `This` types, e.g., `LinkedList.This` vs. `ListItr.This`, but this would complicate the typing rules.

Finally, consider the creation of a new *inner* object on line 29 using `this.new ListItr<ItrI>()`. This expression is type-checked both as a method call (whose receiver is `this`) and as a constructor call. Observe that the bound of `ItrI` is `ReadOnly` (line 27) and the guard on the constructor is `Raw` (line 38), which is legal because a `Raw` constructor can create both mutable and immutable objects.

2.3 OIGJ typing rules

Fig. 4 contains all the OIGJ typing rules. We now discuss each rule. Sec. 3 presents a formal type system based on a simplified version of these rules. Some of the rules are identical to those found in OGJ [33] and IGJ [40] (see Sec. 5 for a comparison with OIGJ).

Ownership nesting Consider the following example:

```
List<This,I,Date<World,I>> l1; // Legal nesting
List<World,I,Date<This,I>> l2; // Illegal!
```

Definition of `l2` has illegal ownership nesting because owned dates might leak, e.g., we can store `l2` in this variable:

```
public static Object<World,ReadOnly> publicAliasToL2;
```

On the one hand, types in OIGJ may have multiple owner parameters, e.g., the type of `l1` has two owner parameters (`This` and `World`). On the other hand, an object may only have a single owner at run time. For example, the type of `l1` will correspond at run time to a list that is owned by `this` while its elements are owned by `World`, and observe that `this` is always inside `World`.

Recall that an owner o_1 is *inside* o_2 iff o_1 is a descendant in the ownership tree of o_2, i.e., $o_1 \preceq_\theta o_2$. We extend this definition from owners to owner parameters as follows: given two owner parameters $\texttt{O}_1$ and $\texttt{O}_2$ in the same type, then $\texttt{O}_1$ is *inside* $\texttt{O}_2$ iff in any possible execution, these owner parameters correspond to some owners o_1 and o_2 (respectively) where $o_1 \preceq_\theta o_2$. For example, `This` is inside `O`, and any owner parameter is inside `World`.

OIGJ requires that owner parameters are properly nested, i.e., that the first owner parameter of type `T` is inside any other owner parameter in `T`. To enforce this rule, OIGJ maintains

Ownership nesting The first owner parameter of type `T` must be inside any other owner parameter in `T`.

Field access Field access `o.f` is legal iff $\underline{O(\texttt{f}) = \texttt{This} \Rightarrow \texttt{o} = \texttt{this}}$.

Field assignment Field assignment `o.f=...` is legal iff (i) $I(\texttt{o}) \leq \texttt{Raw}$, and (ii) $\big(I(\texttt{o}) = \texttt{Raw} \Rightarrow (\texttt{o} = \texttt{this}$ or $O(\texttt{o}) = \texttt{This})\big)$, and (iii) field access `o.f` is legal.

Method invocation Consider method $\texttt{T}_0\ \texttt{m(T}_1\texttt{,...,T}_n\texttt{)}$. The invocation `o.m(...)` is legal iff (i) $\underline{O(\texttt{T}_i) = \texttt{This} \Rightarrow \texttt{o} = \texttt{this}}$ for $i = 0,\ldots,n$, and (ii) $I(\texttt{m}) = \texttt{Raw}$ implies field assignment part (ii).

cJ's method guard (i) An invocation `o.m(...)` is legal if the type of `o` satisfies the guard of `m`. (ii) When typing method `m`, the bound of type variables that appear in the guard changes to their bound in the guard. (iii) The guard of an overriding method is equivalent or weaker than that of the overridden method.

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

Invariant The programmer marks each type parameter as invariant or covariant. An immutability parameter is always covariant, whereas an owner parameter is always invariant.

A type parameter must be invariant if it is used in a superclass that contains `Mutable`, a field that contains `Mutable` but is not `this`-owned, or in the position of another invariant type parameter.

Same-class subtype relation Let $\texttt{C<X}_1\texttt{,...,X}_n\texttt{>}$ be a class. Type $\texttt{S} = \texttt{C<S}_1\texttt{,...,S}_n\texttt{>}$ is *a subtype of* $\texttt{T} = \texttt{C<T}_1\texttt{,...,T}_n\texttt{>}$, written as $\texttt{S} \leq \texttt{T}$, iff $(\texttt{S} = \texttt{T})$ or $\Big($(all immutability parameters $\texttt{T}_j$ are either `ReadOnly` or `Immut`), and for $i = 1,\ldots,n$, $\big(\texttt{S}_i = \texttt{T}_i$ or $(\texttt{S}_i \leq \texttt{T}_i$ and $\texttt{X}_i$ is covariant in `C`$)\big)\Big)$.

Erased signature If method $\texttt{m}'$ overrides a readonly/immutable method `m`, then the erased signatures of $\texttt{m}'$ and `m`, excluding invariant type parameters, must be identical. (The *erased signature* of a method is obtained by replacing type parameters with their bounds.)

Object creation A constructor cannot have any `this`-owned arguments. Furthermore, `new SomeClass<X,...>(...)` is legal iff the constructor's guard `<I extends Y>?` satisfies: `Y` = `Mutable` and `X` = `Mutable`, or `Y` = `Raw`.

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

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

Fresh owners A *fresh owner* is a method owner parameter that is not used in the method signature. It is a descendant in the ownership tree of all other owners in scope.

Static context `This` cannot be used in a static context (static methods or fields).

Figure 4. All the OIGJ typing rules (beyond those of Java), in English. Also see Sec. 3 for a formalization. Underlined sentences show similarities among the rules.

ordering constraints among owner parameters in the same way as described in OGJ [33].

Field access This rule enforces ownership: `this`-owned fields can be assigned only via `this`. In Fig. 3, note that all accesses and assignments to `header` are done via `this`.

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 `Raw` reference. Part (ii) ensures `Raw` is transitive only for `this` or `this`-owned objects. Part (iii) enforces ownership constraints as in field access.

For example, consider the assignments on lines 8 and 9 of Fig. 3. Note that the bound of `I` is `Raw`, thus the assignments satisfy part (i). Part (ii) holds, i.e., `Raw` is transitive in the first assignment because the target object is `this` and in the second assignment because it is `this`-owned (the type of `this.header` is $\texttt{Entry<}\underline{\texttt{This}}\texttt{,I,E>}$). Finally, part (iii) holds in the first assignment because `header` was assigned via `this`, and in the second assignment because field `next` ($\texttt{Entry<}\underline{\texttt{O}}\texttt{,I>}$) is not `this`-owned.

Method invocation Method invocation is handled in the same way as field access/assignment: parts (i) and (ii) are similar to field access and field assignment part (ii). For example, consider the following method: `R m(A a) { ... }` Then, the method call `o.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`. Note that regarding the transitivity of `Raw`, we check both the immutability of the receiver object ($I(\texttt{o})$) and that of the method, i.e., its guard ($I(\texttt{m})$). If both are `Raw`, then we require that `o` is either `this` or `this`-owned.

Inner classes An *inner class* is a non-static nested class, e.g., iterators in `java.util` are implemented using inner classes. An inner class reuses 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 typing rules that involve ownership.

Nested classes that are *static* can be treated the same as normal classes.

Invariant A user can annotate a type parameter `X` in class `C` with `@InVariant` to prevent covariant changes, in which case we say that `X` is invariant. Otherwise we say that `X` is covariant. An immutability parameter must be covariant, or else a mutable reference could not be a receiver when calling a readonly method. An owner parameter must be invariant, because the owner of an object cannot change.

A type parameter must be invariant if it is used in a field/superclass that contains `Mutable`, or if the erased signature differs. For example, if a class has a field of type `Foo<O,Mutable,X>`, then `X` must be invariant (the owner parameter `O` is always invariant).

Subtype relation Java is *invariant* in generic arguments, i.e., it prohibits *covariant* (or contravariant) changes. `Vector<Integer>` is not a subtype of `Vector<Object>`. If it were, then mutating a `Vector<Integer>` 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. OIGJ's subtyping rules includes Java's subtyping rules, therefore OIGJ's subtype relation is a superset of Java's subtype relation. If mutation is disallowed, OIGJ's subtyping rule allows covariant changes in other type parameters, within the *same class*. For example, `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 `Immut`, 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>`.

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 [7]. Such bridge methods work correctly only under the assumptions that subtyping is invariant. For example, consider an integer comparator `intComp` that implements `Comparable<Integer>`. If `Comparable<Integer>` were a subtype of `Comparable<Object>`, then we could pass a `String` to `intComp`'s implementation of `compareTo(Integer)`:
`((Comparable<Object>)intComp).compareTo("a")`

OIGJ requires that the *erased signature* of an overriding method remains the same (excluding invariant parameters) if the overridden method is either readonly or immutable. For example, the erased signature of `compareTo` in `intComp` differs from the one in the interface `Comparable<O,I,X>`. Therefore, this rule requires that the type parameter `X` must be invariant:

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

Object creation A constructor should not have any `this`-owned parameters, because `this`-owned objects can only be created inside `this`.

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 has the guard `<I extends Mutable>?`. Recall that cJ prohibits calling a `Raw` constructor to create an `Immut` object because the guard is not satisfied: `Immut` is not a subtype of `Raw`. OIGJ changed cJ and treats constructor calls using this object creation rule: a `Raw` constructor can create any object (mutable and immutable). A `Mutable` constructor can only create `Mutable` objects. A constructor cannot be `Immut` or `ReadOnly`, so that it is able to assign to the fields of `this`.

Generic wildcards OIGJ uses Java's existing generic wildcard syntax (`?`) to express existential owners [8, 30, 38]. One can use existential owners when the exact owner of an object is unknown. A motivation for existential owners is the downcast performed in the `equals` method [38].

Consider the following two casts in normal Java:

```
boolean equals(Object o)
{  List<?> l = (List<?>)o; // OK
   List<Object> l = (List<Object>)o; } // Warning!
```

The second cast is a warning since erasure makes it impossible to check at run time that the generic parameter is `Object`.

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 a method return type are also prohibited because they can be used to leak `this`-owned fields. However, wildcards on *stack variables* (method parameters or local variables) are allowed.

Note that the immutability parameter is covariant, and therefore there is no need to use a wildcard for immutability. For example, consider the `DateList` class, which is parameterized by its owner parameter (`O`) and the dates' owner parameter (`DO`):

```
class DateList<O,I,DO extends World> {
 boolean equals(Object<?,ReadOnly> o)
 { DateList<?,ReadOnly,?> l =
     // No need to check ownership or immutability at run time.
     (DateList<?,ReadOnly,?>) o;
   return listEquals(l); }
 <O2 extends World,DO2 extends World> boolean
    listEquals(DateList<O2,ReadOnly,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<?,ReadOnly,?>` are now named `DateList<O2,ReadOnly,DO2>`. Although wildcards can sometimes be replaced by generic methods, wildcards are necessary in downcasts (as shown in method `equals`).

Recall that Java's generics can be bypassed by using reflection or raw types such as `List`. Similarly, one can bypass OIGJ when using these features.

`Raw` ***parameter*** `Raw` can only be used after the `extends` keyword. For example, it is prohibited to write `Date<O,Raw>`. If this was possible, then such a date could leak from a `Raw` constructor that is building an immutable object resulting in an alias that could mutate such immutable object.

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. This 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. Hence, 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` wrapper:

```
<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`. In fact, `os` can only be referenced from other stack-local objects, and therefore, when the method returns, `os` can be garbage-collected.

Technically, *a fresh owner is strictly inside all other non-fresh owners in scope*, to make sure it cannot exist after the method returns. (Multiple fresh owners are incomparable with each other.) 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* [37] ownership model, in which the fresh owners are owned 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 `This` represents that an object is owned by `this`, and so OIGJ prohibits using it 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 extending `World`. For example, method:

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

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

2.4 Factory method design pattern

The *factory method pattern* [17] is a creational design pattern for creating objects without specifying the exact class of the object that will be created. The solution is to define an interface with a method for creating an object. Implementers can override the method to create objects of a derived type.

The challenge of the factory method pattern with respect to *ownership* [26] is that the point of *creation* and *usage* are in different classes, and the created object must be owned by its user. Previous work makes a newly-created object be owned by its creator, and then changes the ownership after the fact via sophisticated ownership transfer mechanisms [25] using `capture` and `release`.

In OIGJ's approach, an object has its final owner from its moment of creation. When requesting creation of a new object, the client of the factory also specifies the owner. The type-checker ensures that the created object cannot be *captured* (stored in a location that would require a different owner) in the process. Specifically, a *generic factory method* can abstract over the owner (and immutability) parameter of the constructed object. The underlying generics mechanism finds the correct generic method arguments.

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

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

The documentation of `Collections.synchronizedList` states: "*In order to guarantee serial access, it is* critical *that all access to the backing list is accomplished through the returned list.*" That means that there might be concurrency problems if one accidentally uses the backing list `b` instead of `l`.

So, you want to own a list `l`, which is backed by another list `b`. The challenge is that `b` should be owned by `l` (and not

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

Figure 5. Factory method design pattern in OIGJ. OIGJ guarantees that the backing list `b` (line 5) is never accessed directly, e.g., it cannot be captured on line 19.

by you), in order to guarantee that you do not accidentally access `b` directly and comprise thread-safety.

Fig. 5 shows how owner-as-dominator can ensure that the backing list `b` has no outside aliases. This solution avoids refactoring of existing Java code by delegating calls to the synchronized list `l`. Specifically, class `SafeSyncList` (lines 1–8) owns both the list `l` (line 2) and the backing list `b` (line 5). A factory method is used on lines 3–6.

The `Factory` interface is defined on lines 14–15. 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 statically unknown at the creation point (line 15). The generics mechanism fills in the correct generic arguments from the usage point (line 6) to the actual creation point (line 19). Note that the factory implementation cannot capture an alias to the newly created list on line 19, because its owner parameter `O2` is a generic method parameter that cannot be used in fields.

To conclude, one can use `SafeSyncList` instead of using Sun's unsafe `synchronizedList`, and be certain no one else can access the backing list. All this was achieved using *generic factory methods* on lines 15 and 18.

2.5 Visitor pattern

The *visitor design pattern* [17] 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

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

Figure 6. Visitor pattern in OIGJ. The `Node`'s ownership and immutability are underlined. We omit the `extends` clause for generic parameters, e.g., we assume that `NodeO extends World`. A single visitor interface can be used both for `Mutable` and `ReadOnly` nodes.

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.

Nägeli [26] discusses ownership in design patterns, and shows that previous *ownership* 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 on owner-polymorphic methods: the owner of an object `o`, can pass it to an *owner-polymorphic method*, which cannot capture `o`.

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. 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 14 or mutable on line 22.

Finally, note how the type arguments `This,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 `This,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.

3. Formalization and Type Soundness

Proving soundness is essential in the face of complexities such as wildcards and raw/cooked objects. This section gives the typing rules and operational semantics of a simplified version of OIGJ and sketches the proofs of our immutability and ownership guarantees. For lack of space, the full proofs are included in our technical report [39].

Our type system, called Featherweight OIGJ (FOIGJ), is based on Featherweight Java (FJ) [21]. FOIGJ models the essence of OIGJ: the fact that every object has an ownership and immutability, and the cooking phase when creating immutable objects. FOIGJ adds imperative constructs to FJ, such as `null` values, field assignment, locations/objects, and a heap. FOIGJ also adds a constructor body (to model the cooking phase), owner and immutability parameters to classes, guards as in cJ [20], wildcard owners, and the run-time notion of raw/cooked objects.

FOIGJ poses two main challenges: (i) modeling *wildcards* in the typing rules, and (ii) the representation for *raw objects*. We use the following example (similar to Fig. 2) to demonstrate these two challenges:

```
class Foo<O,I> {
 Date<O,I> sameD;
 Date<This,I> ownedD;
 Date<This,Immut> immutD;
 <I extends Raw>? void Foo(){
   this.ownedD = new Date<This,I>();
   this.immutD = new Date<This,Immut>();
   ... } }
```

Wildcards pose a difficulty due to a process in Java called *wildcard capture* in which a wildcard is replaced with a fresh type variable. For example, the two underlined wildcards below might represent two distinct owners:

```
Foo<?,I> f = ...;
Date<?,I> d = ...;
f.sameD = d; // Illegal assignment! Different owners!
```

A Java compiler rejects the assignment due to incompatible types, because the wildcards were captured by different type variables. Formalizing the full power of wildcards (with upper and lower bounds) was only recently achieved [9]. FOIGJ does *not* model wildcard capture. Instead, it is enough to augment the field assignment rule with the following check: assigning to o is illegal if $O(o) = ?$ (similarly for method invocation). This extra check is needed only in FOIGJ, and not in OIGJ, because OIGJ is built on top of Java, which supports wildcard capture.

The second challenge is modeling raw objects in the *non-erased* operational semantics. Recall that generics are erased in Java and are not present at run time. FOIGJ's *erased* operational semantics is identical to that of normal Java: ownership and immutability information is not kept. In contrast, the *non-erased* version stores with each object its owner and immutability, and it checks at run time the ownership and immutability guarantees (i.e., that field assignment respects owner-as-dominator and is done only on mutable or raw objects). The non-erased version is used only in the formalism. Storing the owner and immutability of every object at run time would be a huge overhead, and is not required for correctness if the program satisfies OIGJ's type rules.

The non-erased semantics of Featherweight Generic Java [21] (FGJ) performs variable substitution for method calls, however FGJ's way of doing substitution does not work in FOIGJ. For example, consider the following reduction as done in imperative FGJ:

```
new Foo<World,Immut>() →
                    l.ownedD = new Date<l,Immut>();
                    l.immutD = new Date<l,Immut>();...
```

The variable `I` in the constructor was substituted with `Immut`, and the variables `this` and `This` were substituted with a new location `l` that was created on the heap, i.e., the heap H now contains a new object in location `l` whose fields are all `null`: $H = \{\mathtt{l} \mapsto \mathtt{Foo<World,Immut>(\overline{null})}\}$. (Locations are pointers to objects; we treat locations and objects identically because they have a one-to-one mapping, e.g., the owner of a location is defined to be the owner of its object.) Note how owner parameters ($O(o)$) at compile time are replaced with owners ($\theta(o)$) at run time, e.g., `This` was replaced with location `l`.

There are two reasons why substituting `I` with `Immut` does not work in FOIGJ: (i) the reduction does not type-check because we mutate an immutable object (`l.ownedD = ...`), and (ii) we lost information about the two new `Date` objects, namely that `ownedD` can still be mutated after its constructor finishes (because it is `this`-owned) whereas `immutD` cannot.

FOIGJ solves these two issues by introducing an auxiliary type $\mathtt{Immut}_{\mathtt{l}}$. An object o of immutability $I(o) = \mathtt{Immut}_{\mathtt{l}}$ becomes cooked when the constructor of `l` finishes, therefore we call `l` its *cooker*, denoted by $\kappa(o) = \mathtt{l}$. Phrased differently, an object is cooked when its cooker is *constructed* (i.e., the cooker's constructor finishes). Note that the cooker `l` can be o itself, its owner, or even some other incomparable object.

The connection between the cooker and the owner will be shown in the subtyping and typing rules below. Intuitively, for a reference of type $\mathtt{C<o,Immut}_{\mathtt{l}}\mathtt{>}$, if the cooker `l` is not inside the owner `o`, then that reference must point to an object whose cooker is `l`. Otherwise (if `l` is inside `o`), then that reference might point to any cooked immutable object (even one with a cooker that is not `l`).

In our example, the location `l` that was created with `new Foo<World,Immut>()` becomes cooked when it is constructed, i.e., $\kappa(\mathtt{l}) = \underline{\mathtt{l}}$, and $H = \{\mathtt{l} \mapsto \mathtt{Foo<World,Immut}_{\underline{\mathtt{l}}}\mathtt{>(\overline{null})}\}$. Now FGJ's way of doing the substitution works for FOIGJ, because `I` is replaced with $\mathtt{Immut}_{\mathtt{l}}$, i.e.,

```
l.ownedD = new Date<l,Immut_l>();
l.immutD = new Date<l,Immut>();
```

Note how the cooker of `ownedD` is `l`, whereas the cooker of `immutD` is `immutD` itself. Therefore, `ownedD` has a longer cooking phase than `immutD`.

FOIGJ also maintains the set of currently executing constructors K, where $K \subseteq \mathrm{dom}(H)$. We maintain the invariant that a location `l` is *raw* iff $\kappa(\mathtt{l}) \in K$, and require that *only mutable or raw objects can be mutated*. Specifically, $\mathtt{Immut}_{\mathtt{l}}$ is a subtype of `Raw` when $\mathtt{l} \in K$, and it is a subtype of `Immut` when $\mathtt{l} \notin K$.

Type $\mathtt{Immut}_{\mathtt{l}}$ also helps understand the **Object creation rule** better. Recall that an `Immut` object can be created from a `Raw` constructor, even though `Immut` is not a subtype of `Raw`, which seems to contradict **cJ's method guard rule**. However, type $\mathtt{Immut}_{\mathtt{l}}$ is in fact a subtype of `Raw` when the object is created, because in our typing rules we have that $\mathtt{Immut}_{\mathtt{l}} \leq \mathtt{Raw}$ iff $\mathtt{l} \in K$, and when an object is created, its cooker must be in K. Phrased differently, the type of an immutable object never changes (always $\mathtt{Immut}_{\mathtt{l}}$), but during the program execution the set K changes, and therefore the subtyping relation changes: initially $\mathtt{Immut}_{\mathtt{l}}$ is a subtype of `Raw`, but later the object becomes cooked, and then $\mathtt{Immut}_{\mathtt{l}}$ is no longer a subtype of `Raw`, but instead it becomes a subtype of `Immut`.

Faced with such major challenges, we removed from FOIGJ anything that was not needed to prove our run-time guarantees. Specifically, FOIGJ does *not* model: generics (except for the owner and immutability parameters), owner polymorphic methods, casting, inner classes, fresh owners, or multiple immutability/owner parameters. On the one hand, the interaction between generics and *immutability* (which enables covariant subtyping) was previously proven sound in Featherweight IGJ (FIGJ) [40]. On the other hand, the interaction between generics and *ownership* (as found in the ownership nesting rule) was previously proven sound in Featherweight OGJ (FOGJ) [33]. Because covariant subtyping (as found in IGJ) and ownership nesting (as found in OGJ) was not changed in OIGJ, we decided not to model generics in FOIGJ. We note that FOIGJ does model `Raw`, which was not modeled previously in FIGJ.

Consider the typing rules in Fig. 4. Classes in FOIGJ have a single `Raw` constructor, therefore **Object creation rule** is always satisfied and can be ignored. Furthermore, because FOIGJ does not model generics, static, or inner classes, then the following rules can also be ignored: **Ownership nesting**, **Inner classes**, **Inheritance**, **Invariant**, **Erased signature**, and **Fresh owners**. Covariant subtyping and erased signatures were described in FIGJ, and ownership nesting and

FT ::= C<FO,IP>	Field (and method return) Type.
T ::= C<MO,IP>	Type.
N ::= C<NO,NI>	Non-variable type (for objects).
NO ::= World | l	Non-variable Owner parameter (for objects).
FO ::= NO | This | O	Field Owner parameter.
MO ::= FO | ?	Method Owner parameter (including generic wildcard).
NI ::= Mutable | Immut$_{\texttt{l}}$	Non-variable Immutability parameter (for objects).
VI ::= NI | Immut | I	Variable Immutability for new.
IP ::= ReadOnly | VI	Immutability Parameter.
IG ::= ReadOnly | Immut | Mutable | Raw	Immutability method Guard.
M ::= <I extends IG>? FT m($\overline{\texttt{T}}\ \overline{\texttt{x}}$) { return e; }	Method declaration.
L ::= class C<O,I> extends C′<O,I>{ $\overline{\texttt{FT}}\ \overline{\texttt{f}}$;$\overline{\texttt{M}}$ }	cLass declaration.
v ::= null | l	Values: either null or a location l.
e ::= v | x | e.f | e.f = e | e.m($\overline{\texttt{e}}$) | new C<FO,VI>($\overline{\texttt{e}}$) | e ;return l	Expressions.

Figure 7. FOIGJ Syntax. The terminals are null, owner parameters (O, This, World), and immutability parameters (I, ReadOnly, Mutable, Raw, Immut). Given a location l, Immut$_{\texttt{l}}$ represents an immutable object with cooker l. The program source code cannot contain the grayed elements (locations are only created during execution/reduction in R-NEW of Fig. 10).

(limited) owner-polymorphic methods in FOGJ. We stress that FOIGJ *does* model wildcard for the owner parameter (?), which is used in owner-polymorphic methods such as sort or equals. In our view, extending the formalism with fresh owners or inner classes increases the complexity of the calculus without providing new insights.

The following rules are enforced by the syntax of FOIGJ (Fig. 7): **Generic Wildcards** and **Raw parameter**. The remaining rules are: **Field assignment**, **Field access**, **Method invocation**, **cJ's [20] method guard**, and a **Subtype relation** (without generics). These rules are formalized in FOIGJ in the subtyping rules of Fig. 8 ($K,\Gamma \vdash \texttt{T} \leq \texttt{T}'$) and the typing rules of Fig. 9 ($K,\Gamma \vdash \texttt{e} : \texttt{T}$). Finally, the reduction rules are described in Fig. 10 ($K \vdash H,\texttt{e} \rightarrow H',\texttt{e}'$).

Sec. 3.1 describes the syntax of FOIGJ, Sec. 3.2 the subtyping rules, Sec. 3.3 the typing rules, Sec. 3.4 the reduction rules, and Sec. 3.5 proves preservation, progress, and our run-time immutability and ownership guarantees.

3.1 Syntax of FOIGJ

FOIGJ adds imperative extensions to FJ such as assignment to fields, object locations, null, and a heap [32]. A constructor initializes all the fields to null, and then calls a build method that constructs the object. Having null values is important because this-owned fields must be initialized with null since they cannot be assigned from the outside, i.e., they must be created within this. For example, a list constructor cannot receive its entries as constructor arguments; instead it must create the entries within the build method.

Fig. 7 presents the syntax of FOIGJ. Expressions in FOIGJ include the four expressions in FJ (method parameter, field access, method invocation, and new instance creation; *without* casting), as well as the imperative extensions (field update, e;return l, and values). Expression e;return l is created when reducing a constructor call, e.g., $K \vdash$ new N(...) $\rightarrow$ l.build(...);return l, then we proceed to reduce l.build(...) and finally return l. Note that O and I are terminals, i.e., the owner and immutability parameters are always named O and I.

An evaluation of an expression (e) is either infinite, or is stuck on null-pointer exception, or terminates with a value (v), which is either null or a location l.

Note how the syntax limits the usage of wildcards and Raw: wildcards (?) can be used only as the owner of method arguments (FOIGJ does not have local variables), and Raw only as a method guard (IG).

We represent sequences using an over-line notation, similarly to FJ, i.e., comma denotes concatenation of sequences, and $\overline{\texttt{FT}}\ \overline{\texttt{f}}$; represents the sequence $\texttt{FT}_1\ \texttt{f}_1; \ldots \texttt{FT}_n\ \texttt{f}_n;$

A class in FOIGJ has a single constructor that can create both mutable and immutable objects, i.e., it is a Raw constructor. The constructor is not shown in the syntax because it can be inferred from the class declaration: it always assigns null to the fields of the newly created object, and then invokes this special method (ignoring the return value):

```
<I extends Raw>? T′ build(T̄ ē) { return e; }
```

We require that each class has such a method, and that its parameters are not this-owned nor have wildcards. The reduction rules call that method after the fields are set to null.

3.2 Subtyping in FOIGJ

An environment Γ is a finite mapping from variables x and locations l to types T, e.g., x : T $\in \Gamma$. The location types define the ownership tree $\preceq_\theta$ (or without reflexivity $\prec_\theta$). The set of currently executing constructors is denoted K. In addition, Γ maps the immutability parameter I to its bound according to the current method's guard (IG in Fig. 7). For

$$\frac{}{K,\Gamma \vdash \mathtt{I} \leq \Gamma(\mathtt{I})}\ (\mathrm{S1}) \qquad \frac{}{K,\Gamma \vdash \mathtt{T} \leq \mathtt{T}}\ (\mathrm{S2}) \qquad \frac{K,\Gamma \vdash \mathtt{S} \leq \mathtt{T} \quad K,\Gamma \vdash \mathtt{T} \leq \mathtt{U}}{K,\Gamma \vdash \mathtt{S} \leq \mathtt{U}}\ (\mathrm{S3}) \qquad \frac{\mathtt{class\ C<O,I>\ extends\ C'<O,I>}}{K,\Gamma \vdash \mathtt{C<MO,IP>} \leq \mathtt{C'<MO,IP>}}\ (\mathrm{S4})$$

$$\frac{}{K,\Gamma \vdash \mathtt{Mutable} \leq \mathtt{Raw}}\ (\mathrm{S5}) \qquad \frac{}{K,\Gamma \vdash \mathtt{Raw} \leq \mathtt{ReadOnly}}\ (\mathrm{S6}) \qquad \frac{}{K,\Gamma \vdash \mathtt{Immut} \leq \mathtt{ReadOnly}}\ (\mathrm{S7})$$

$$\frac{K,\Gamma \vdash \mathtt{IP} \leq \mathtt{IP'}}{K,\Gamma \vdash \mathtt{C<MO,IP>} \leq \mathtt{C<MO,IP'>}}\ (\mathrm{S8}) \qquad \frac{}{K,\Gamma \vdash \mathtt{C<MO,IP>} \leq \mathtt{C<?,IP>}}\ (\mathrm{S9}) \qquad \frac{\mathtt{l} \in K}{K,\Gamma \vdash \mathtt{Immut_l} \leq \mathtt{Raw}}\ (\mathrm{S10})$$

$$\frac{\mathtt{l} \notin K}{K,\Gamma \vdash \mathtt{Immut_l} \leq \mathtt{Immut}}\ (\mathrm{S11}) \qquad \frac{\mathtt{l} \notin K}{K,\Gamma \vdash \mathtt{Immut} \leq \mathtt{Immut_l}}\ (\mathrm{S12}) \qquad \frac{\mathtt{l} \prec_\theta \mathtt{NO}}{K,\Gamma \vdash \mathtt{C<NO,Immut>} \leq \mathtt{C<NO,Immut_l>}}\ (\mathrm{S13})$$

Figure 8. FOIGJ Subtyping Rules ($K,\Gamma \vdash \mathtt{T} \leq \mathtt{T'}$). Rule S13 shows the connection between cooker `l` and owner `NO`.

example, $\mathtt{I} : \mathtt{Raw} \in \Gamma$ when typing the expression `e` in method `build` above.

Fig. 8 shows FOIGJ subtyping rules. Rules S1–S4 are the same as FGJ rules: S1 means that a generic variable is a subtype of its bound, S2 is reflexivity, S3 is transitivity, and S4 is that subclassing defines subtyping. Rules S5–S7 show subtyping among non-variable immutability parameters as shown in Fig. 1b. Rule S8 defines covariant subtyping for the immutability parameter. Rule S9 formalizes subtyping with a wildcard owner.

The last four rules S10–S13 are concerned with *cookers* such as $\mathtt{Immut_l}$. Recall that an object is cooked when its cooker `l` is constructed, i.e., the constructor of `l` is no longer executing: $\mathtt{l} \notin K$. Rule S10 views the type as `Raw`, while rules S11–S12 shows the equivalence to `Immut`. Note that subtyping is no longer antisymmetric, i.e., there are non-equal types $\mathtt{T}_1$ and $\mathtt{T}_2$ for which $\mathtt{T}_1 \leq \mathtt{T}_2 \leq \mathtt{T}_1$. For example, $\mathtt{T}_1 = \mathtt{C<O,Immut_l>}$ and $\mathtt{T}_2 = \mathtt{C<O,Immut>}$, when $\mathtt{l} \notin K$. In fact, this is not surprising because these types both represent immutable object, and after the cooker is cooked, the identity of the cooker is irrelevant.

Cooker vs. owner Rule S13 assumes that the cooker is inside the owner ($\mathtt{l} \prec_\theta \mathtt{NO}$), which means the object might came from the outside. This rule addresses the difference between the cooker of (i) a location `l` or (ii) that of an expression such as field access `l.f`: (i) location `l` will be cooked *exactly* when the constructor of $\kappa(\mathtt{l})$ is finished, however, (ii) the cooker of `l.f` is an *over-approximation*, i.e., the object stored in that field might have been cooked earlier. Rule S13 allows an over-approximation only when the cooker is inside the owner.

Consider this example:

```
class Foo<O,I> { Date<O,I> same;
 <I extends Raw>? Foo(Date<O,I> d) { same=d; }  }
```

Field `same` is assigned from the outside, but it might still be `this`-owned. We will show the reduction of two expressions: one where `same` is assigned a cooked (outside) date, and one with a raw date. The expressions are inside the constructor of some object `b` whose cooker is `b` itself.

The reduction of the first expression:

```
new Foo<This,Immut>(new Date<This,Immut>())
```

results in the heap: $H = \{\mathtt{d1} \mapsto \mathtt{Date<b,Immut_{d1}>()}, \mathtt{f1} \mapsto \mathtt{Foo<b,Immut_{f1}>(d1)}\}$. Note that the type of `d1` must be a subtype of `f1.same` in a well-typed heap (formally defined later). The type of `d1` is a subtype of `Date<b,Immut>` (because $\mathtt{d1} \notin K$ in rule S12), which is a subtype of $\mathtt{Date<b,Immut_{f1}>}$ (because $\mathtt{f1} \prec_\theta \mathtt{b}$ in rule S13). Phrased differently, the cooker of `f1.same` is `f1`, but it may point to an object that was cooked before, and indeed it points to an object whose cooker is `d1` (so it is an over-approximation).

The reduction of the second expression:

```
new Foo<This,I>(new Date<This,I>())
```

results in the heap (because $\mathtt{I} = \mathtt{Immut_b}$): $H = \{\mathtt{d2} \mapsto \mathtt{Date<b,Immut_b>()}, \mathtt{f2} \mapsto \mathtt{Foo<b,Immut_b>(d2)}\}$. Note that in this case, both `d2` and `f2` have the same cooker `b`. The type of `f2.same` is $\mathtt{Date<b,Immut_b>}$, and because $\mathtt{b} \not\prec_\theta \mathtt{b}$ (see rule S13), then we know that this is not an over-approximation, i.e., that field points to an object whose cooker must be `b`.

To summarize, consider a type $\mathtt{Foo<o,Immut_c>}$. If the cooker `c` is inside the owner `o` ($\mathtt{c} \prec_\theta \mathtt{o}$), or the cooker is cooked ($\mathtt{c} \notin K$), then the type is an over-approximation, i.e., it can point to any `Immut` object (that is, to any object with cooker $\mathtt{c'} \notin K$). Otherwise, it points to an object whose cooker is exactly `c`. Formally,

LEMMA 3.1. *If* $K,\Gamma \vdash \mathtt{C<MO,IP>} \leq \mathtt{C'<NO,Immut_l>}$, $\mathtt{l} \not\prec_\theta \mathtt{NO}$, *and* $\mathtt{l} \in K$, *then* $\mathtt{IP} = \mathtt{Immut_l}$.

We also prove in the technical report that:

LEMMA 3.2. *If* $K,\Gamma \vdash \mathtt{C<MO,IP>} \leq \mathtt{C'<MO',IP'>}$, *then (i)* $\mathtt{MO'} \neq \mathtt{?} \Rightarrow \mathtt{MO} = \mathtt{MO'}$, *(ii)* $(\mathtt{IP'} \neq \mathtt{Immut_l}$ *or* $\mathtt{l} \not\prec_\theta \mathtt{MO'}) \Rightarrow K,\Gamma \vdash \mathtt{IP} \leq \mathtt{IP'}$, *and (iii)* `C` *is a subclass of* `C'`, *(iv)* $K,\Gamma \vdash \mathtt{D<l,IP>} \leq \mathtt{D<l,IP'>}$ *for any class* `D` *and location* `l` *where* $\mathtt{MO'} \preceq_\theta \mathtt{l}$.

3.3 Typing rules of FOIGJ

Auxiliary functions We use the following auxiliary functions: *fields*(`C`) returns all the field names (including inherited fields) of class `C`, *ftype*(`f`,`C`) returns the type of field `f` in class `C`, *mtype*(`m`,`C`) returns the type of method `m` in class `C`, and *mbody*(`m`,`C`) returns its body. Their definitions are based on their counterparts in FJ, and thus omitted from this paper. In addition, function *mguard*(`m`,`C`) returns the method's guard (`IG` in Fig. 7).

$$\frac{K \cup \{\texttt{l}\}, \Gamma \vdash \texttt{e} : \texttt{T}}{K, \Gamma \vdash \texttt{e;return l} : \Gamma(\texttt{l})}\ \text{(T-RETURN)} \qquad \frac{\textit{mtype}(\bot, \texttt{build}, \texttt{C<FO,VI>}) = \overline{\texttt{T}} \rightarrow \texttt{U} \quad K, \Gamma \vdash \overline{\texttt{e}} : \overline{\texttt{T}'} \quad K, \Gamma \vdash \overline{\texttt{T}'} \leq \overline{\texttt{T}}}{K, \Gamma \vdash \texttt{new C<FO,VI>}(\overline{\texttt{e}}) : \texttt{C<FO,VI>}}\ \text{(T-NEW)}$$

$$\frac{}{K, \Gamma \vdash \texttt{x} : \Gamma(\texttt{x})}\ \text{(T-VAR)} \qquad \frac{}{K, \Gamma \vdash \texttt{null} : \texttt{T}}\ \text{(T-NULL)} \qquad \frac{K, \Gamma \vdash \texttt{e} : \texttt{C<MO,IP>} \quad \textit{ftype}(\texttt{e}, \texttt{f}, \texttt{C<MO,IP>}) = \texttt{T}}{K, \Gamma \vdash \texttt{e.f} : \texttt{T}}\ \text{(T-FIELD-ACCESS)}$$

$$\frac{}{K, \Gamma \vdash \texttt{l} : \Gamma(\texttt{l})}\ \text{(T-LOCATION)} \qquad \frac{\begin{array}{c} K, \Gamma \vdash \texttt{e.f} : \texttt{T} \quad K, \Gamma \vdash \texttt{e}' : \texttt{T}' \quad K, \Gamma \vdash \texttt{T}' \leq \texttt{T} \quad K, \Gamma \vdash \texttt{e} : \texttt{C<MO,IP>} \\ K, \Gamma \vdash \texttt{IP} \leq \texttt{Raw} \quad \textit{isTransitive}(\texttt{e}, \Gamma, \texttt{C<MO,IP>}) \quad \texttt{MO} \neq \texttt{?} \end{array}}{K, \Gamma \vdash \texttt{e.f} = \texttt{e}' : \texttt{T}'}\ \text{(T-FIELD-ASSIGNMENT)}$$

$$\frac{\begin{array}{c} K, \Gamma \vdash \texttt{e}_0 : \texttt{C<MO,IP>} \quad \textit{mtype}(\texttt{e}_0, \texttt{m}, \texttt{C<MO,IP>}) = \overline{\texttt{T}} \rightarrow \texttt{T"} \quad K, \Gamma \vdash \overline{\texttt{e}} : \overline{\texttt{T}'} \quad K, \Gamma \vdash \overline{\texttt{T}'} \leq \overline{\texttt{T}} \quad \textit{mguard}(\texttt{m}, \texttt{C}) = \texttt{IG} \\ K, \Gamma \vdash \texttt{IP} \leq \texttt{IG} \quad \texttt{IG} = \texttt{Raw} \Rightarrow \textit{isTransitive}(\texttt{e}_0, \Gamma, \texttt{C<MO,IP>}) \quad \textit{mtype}(\texttt{m}, \texttt{C}) = \overline{\texttt{U}} \rightarrow \texttt{V} \quad O(\texttt{T}_i) = \texttt{?} \Rightarrow O(\texttt{U}_i) = \texttt{?} \end{array}}{K, \Gamma \vdash \texttt{e}_0\texttt{.m}(\overline{\texttt{e}}) : \texttt{T"}}\ \text{(T-INVOKE)}$$

Figure 9. FOIGJ Expression Typing Rules ($K, \Gamma \vdash \texttt{e} : \texttt{T}$).

We overload the auxiliary functions above to work also for types ($\texttt{T} = \texttt{C<MO,IP>}$) and not just classes (`C`) by substituting [`MO/O,IP/I`]. However, we also need to carefully substitute `This` when the receiver is `this` or locations. Function $\textit{ftype}(\underline{\texttt{e}}, \texttt{f}, \texttt{T})$ returns the type of the field access $\underline{\texttt{e}}\texttt{.f}$ where `T` is the type of `e`, or `error` if the access is illegal. For example,

$$\begin{aligned} \textit{ftype}(\texttt{ownedD,Foo}) &= \texttt{Date<This,I>} \\ \textit{ftype}(\texttt{this,ownedD,Foo<O,Immut>}) &= \texttt{Date<This,Immut>} \\ \textit{ftype}(\texttt{this.f,ownedD,Foo<O,Immut>}) &= \texttt{error} \\ \textit{ftype}(\texttt{l,ownedD,Foo<o,Immut}_\texttt{c}\texttt{>}) &= \texttt{Date<l,Immut}_\texttt{c}\texttt{>} \end{aligned}$$

Formally, $\textit{ftype}(\texttt{e,f,C<MO,IP>}) = [\texttt{MO/O,IP/I,z/This}]\textit{ftype}(\texttt{f,C})$, where (i) $\texttt{z} = \texttt{l}$ if $\texttt{e} = \texttt{l}$, (ii) $\texttt{z} = \texttt{This}$ if $\texttt{e} = \texttt{this}$, (iii) otherwise $\texttt{z} = \texttt{error}$ (and if a type contains `error` then it means the call to *ftype* failed). When we know the field is not `this`-owned, then the expression `e` is not used, and we write $\textit{ftype}(\bot, \texttt{f}, \texttt{C})$. However, if the field is `this`-owned, then `e` must be `this` or a location `l`.

Recall that **Field access rule** in Fig. 4 required that `this`-owned fields can be accessed only via `this`. At run time, `this` is substituted with a location `l`. Therefore, there is a *duality* between `this` and a location `l` in the definition of *ftype*. For example, the field access $\underline{\texttt{this}}\texttt{.ownedD}$ of type $\texttt{Date<}\underline{\texttt{This}}\texttt{,I>}$ is legal because we accessed a `this`-owned field via $\underline{\texttt{this}}$. At run time, `this` is substituted with some location `l`, and the access $\underline{\texttt{l}}\texttt{.ownedD}$ of type $\texttt{Date<}\underline{\texttt{l}}\texttt{,...>}$ is now still legal because we accessed a `this`-owned field via $\underline{\text{a location } \texttt{l}}$. Note that if the access is not done via a location, e.g., `bar.l.ownedD`, then we cannot type-check the resulting expression (because we do not know what should be the substitute for `This`).

There is a similar duality in **Field assignment rule** part (ii), that checks that `Raw` is transitive for `this` or `this`-owned objects. The dual of `this` is a location `l`, and the dual of a `this`-owned object (`C<This,I>`) is an object whose cooker is not inside its owner ($\texttt{C<o,Immut}_\texttt{l}\texttt{>}$ where $\texttt{l} \not\preceq_\theta \texttt{o}$). The second duality holds because, for type `C<This,I>`, the cooker (`I`) is never inside the owner (`This`). At run time, the owner will be the location of `this`, and the cooker is either `this` or some other object that was created before `this`, i.e., the cooker is never inside the owner (but they might be equal).

Function *isTransitive* checks whether `Raw` is transitive. The underlined part shows the *dual* version of **Field assignment rule** part (ii): (i) `this` vs. $\texttt{l}'$, and (ii) $\texttt{MO} = \texttt{This}$ vs. $\texttt{l} \not\preceq_\theta \texttt{MO}$.

$$\begin{aligned} &\textit{isTransitive}(\texttt{e}, \Gamma, \texttt{C<MO,IP>}) = \\ &\quad \big(\texttt{IP} = \texttt{I} \text{ and } \Gamma(\texttt{I}) = \texttt{Raw} \Rightarrow (\texttt{e=this} \text{ or } \texttt{MO} = \texttt{This})\big) \text{ or} \\ &\quad \big(\underline{\texttt{IP} = \texttt{Immut}_\texttt{l} \Rightarrow (\texttt{e=l}' \text{ or } \texttt{l} \not\preceq_\theta \texttt{MO})}\big) \end{aligned}$$

Typing class declarations FOIGJ program consists of class declarations followed by the program's expression. Next, we describe in words the rules for typing the class declarations, and the rules for typing an expression are given formally in Fig. 9. When typing an expression, we assumed the class declarations are well-formed.

To check that class declarations are well-formed, FOIGJ first performs all the checks done in FJ, e.g., that there are no cycles in the inheritance relation, that field and method names are unique in a class, that `this` is not a legal method parameter name, that an overriding method maintains the same signature, etc. FOIGJ performs additional checks related to method guards when typing method declarations, i.e., we modify rule T-METHOD in FJ as follows: (i) An overriding method can only make the guard weaker, i.e., if a method with guard `IG` overrides one with guard $\texttt{IG}'$ then $\texttt{IG}' \leq \texttt{IG}$. (ii) In class `C`, when typing a method: $\texttt{<I extends IG>? FT m}(\overline{\texttt{T}}\ \overline{\texttt{x}})\ \texttt{\{ return e; \}}$ we use an environment Γ in which the bound of `I` is `IG`, i.e., $\Gamma = \{\texttt{I} : \texttt{IG}, \overline{\texttt{x}} : \overline{\texttt{T}}, \texttt{this} : \texttt{C<O,I>}\}$, and we must prove that $\emptyset, \Gamma \vdash \texttt{e} : \texttt{S}$ and $\emptyset, \Gamma \vdash \texttt{S} \leq \texttt{FT}$. Finally, we require that if $\texttt{IG} = \texttt{ReadOnly}$ then $I(\texttt{T}_i) \neq \texttt{I}$. This last requirement is not really a limitation, because a programmer can replace `I` with `ReadOnly` for parameters in readonly methods, and previously legal programs would remain legal. (This requirement is needed to prove preservation for the congruence rule of method receiver, see our technical report for details.)

$$\frac{\mathtt{l} \notin \mathrm{dom}(H) \qquad \mathtt{VI}' = \begin{cases} \mathtt{Immut}_{\mathtt{l}} & \text{if } \mathtt{VI} = \mathtt{Immut} \text{ or } (\mathtt{VI} = \mathtt{Immut}_{\mathtt{c}} \text{ and } \mathtt{c} \notin K) \\ \mathtt{VI} & \text{otherwise} \end{cases}}{K \vdash H, \mathtt{new\ C<NO,VI>}(\overline{\mathtt{v}}) \to H[\mathtt{l} \mapsto \mathtt{C<NO,VI'>}(\overline{\mathtt{null}})], \mathtt{l.build}(\overline{\mathtt{v}})\mathtt{;return\ l}} \ (\text{R-NEW})$$

$$\frac{K \cup \{\mathtt{l}\} \vdash H, \mathtt{e} \to H', \mathtt{e}'}{K \vdash H, \mathtt{e;return\ l} \to H', \mathtt{e}'\mathtt{;return\ l}} \ (\text{R-c1}) \qquad \frac{H[\mathtt{l}] = \mathtt{C<NO,NI>}(\overline{\mathtt{v}}) \qquad \mathit{fields}(\mathtt{C}) = \overline{\mathtt{f}}}{K \vdash H, \mathtt{l.f}_i \to H, \mathtt{v}_i} \ (\text{R-FIELD-ACCESS})$$

$$\frac{H[\mathtt{l}] = \mathtt{C<NO,NI>}(\overline{\mathtt{v}}) \qquad \mathit{fields}(\mathtt{C}) = \overline{\mathtt{f}} \qquad \mathtt{NI} = \mathtt{Mutable} \text{ or } \kappa_H(\mathtt{l}) \in K \qquad \mathtt{v}' = \mathtt{null} \text{ or } \mathtt{l} \preceq_\theta \theta_H(\mathtt{v}')}{K \vdash H, \mathtt{l.f}_i = \mathtt{v}' \to H[\mathtt{l} \mapsto \mathtt{C<NO,NI>}([\mathtt{v}'/\mathtt{v}_i]\overline{\mathtt{v}})], \mathtt{v}'} \ (\text{R-FIELD-ASSIGNMENT})$$

$$\frac{}{K \vdash H, \mathtt{v;return\ l} \to H, \mathtt{l}} \ (\text{R-RETURN}) \qquad \frac{H[\mathtt{l}] = \mathtt{C<NO,NI>}(\ldots) \qquad \mathit{mbody}(\mathtt{m}, \mathtt{C}) = \overline{\mathtt{x}}.\mathtt{e}'}{K \vdash H, \mathtt{l.m}(\overline{\mathtt{v}}) \to H, [\overline{\mathtt{v}}/\overline{\mathtt{x}}, \mathtt{l/this}, \mathtt{l/This}, \mathtt{NO/O}, \mathtt{NI/I}]\mathtt{e}'} \ (\text{R-INVOKE})$$

Figure 10. FOIGJ Reduction Rules ($K \vdash H, \mathtt{e} \to H', \mathtt{e}'$), excluding all congruence rules except R-c1.

Typing expressions Fig. 9 shows the typing rules for expressions in FOIGJ. Most of these rules are a direct translation from Fig. 4. The main challenge was handling *wildcards* correctly without resulting in wildcard-capture. Rule T-FIELD-ASSIGNMENT requires that $\mathtt{MO} \neq \mathtt{?}$, i.e., one cannot assign to an object with unknown owner. Typing method parameters is similar to typing field-assignment, however, method parameters can have a wildcard owner whereas fields cannot (see the difference between `T` and `FT` in Fig. 7). Thus, rule T-INVOKE requires that $O(\overline{\mathtt{T}}) = \mathtt{?} \Rightarrow O(\overline{\mathtt{U}}) = \mathtt{?}$, i.e., if $\mathtt{T}_i = \mathtt{C<?,IP>}$ then $\mathtt{U}_i = \mathtt{C<?,IP'>}$. I.e., if the owner of $\mathtt{e}_0$ is unknown, then the owner of the method parameters cannot be `O`.

Rule T-NEW performs less checks compared to a method call (e.g., no need to check the guard, *isTransitive*, nor wildcards) because `build` has several restrictions: its guard is `Raw` and it does not contain wildcards nor `this`-owned parameters. Because `This` does not appear in the signature of `build`, we know that *mtype* will not use $\perp$ in the call: $\mathit{mtype}(\perp, \mathtt{build}, \mathtt{C<FO,VI>})$.

An expression/type is called *closed* if it does not contain any free variables (e.g., wildcards, `this`, `I`, `O`, or `This`), but it may contain `World`, `ReadOnly`, `Mutable`, `Immut`, $\mathtt{Immut}_{\mathtt{l}}$ or `l`. Note that the type of a closed expression is also closed.

LEMMA 3.3. *If $K, \Gamma \vdash \mathtt{e}' : \mathtt{T}'$ and $\mathtt{e}'$ is closed and $\mathtt{e}' \neq \mathtt{null}$, then $\mathtt{T}'$ is closed.*

The delicate part of the proof is showing that $\mathtt{T}'$ does not contain `This`. Note that *ftype* returns a type with `This` only if $\mathtt{e} = \mathtt{this}$ (which cannot happen since `e` is closed).

3.4 Reduction rules of FOIGJ

The initial expression to be reduced is *closed*, and we guarantee that a closed expression is always reduced to another closed expression:

LEMMA 3.4. *If `e` is closed and $K \vdash H, \mathtt{e} \to H', \mathtt{e}'$, then $\mathtt{e}'$ is closed.*

The heap (or store) H maps each location `l` to an object $\mathtt{C<NO,NI>}(\overline{\mathtt{v}})$, where $\theta_H(\mathtt{l}) = \mathtt{NO}$ is its owner, and $I_H(\mathtt{l}) = \mathtt{NI}$ is its immutability, and $\overline{\mathtt{v}}$ are the values of its fields. If $\mathtt{NI} = \mathtt{Immut}_{\mathtt{l}'}$ then we say that its cooker is $\kappa_H(\mathtt{l}) = \mathtt{l}'$. (We added the subscript H to the functions that return the owner, immutability and cooker, in order to explicitly show the dependence on the heap.) We define a *heap-typing* $\Gamma_H : \mathtt{l} \mapsto \mathtt{T}$ that gives a type to each location in the obvious way (simply removing the list of fields $(\overline{\mathtt{v}})$).

The set of currently executing constructors is K. A heap H is *well-typed for K* if it satisfies two conditions: (i) Each field location is a subtype (using K, Γ_H) of the declared field type, i.e., for every location `l`, where $H[\mathtt{l}] = \mathtt{C<NO,NI>}(\overline{\mathtt{v}})$ and $\mathit{fields}(\mathtt{C}) = \overline{\mathtt{f}}$, and for every field $\mathtt{f}_i$, we have that either $\mathtt{v}_i = \mathtt{null}$ or $K, \Gamma_H \vdash \Gamma_H(\mathtt{v}_i) \leq \mathit{ftype}(\mathtt{l}, \mathtt{f}_i, \mathtt{C<NO,NI>})$. (ii) There is a linear order $\preceq^T$ over $\mathrm{dom}(H)$ such that for every location `l`, $\theta_H(\mathtt{l}) = \mathtt{World}$ or $\theta_H(\mathtt{l}) \prec^T \mathtt{l}$, and $I_H(\mathtt{l}) = \mathtt{Mutable}$ or $\kappa_H(\mathtt{l}) \preceq^T \mathtt{l}$. The linear order $\preceq^T$ can order the objects according to their *creation time*, because $\theta_H(\mathtt{l})$ is always created before `l`, and $\kappa_H(\mathtt{l})$ is either `l` or created before `l`.

In our technical report we prove that if H is well-typed for K then (a) owner-as-dominator holds (Lem. 3.5), and (b) H is well-typed for any subset of K (Lem. 3.6). Part (b) is not trivial, because the subtyping relation for a subset of K is different because raw objects become immutable. Intuitively, during execution objects become cooked (when their cooker is removed from K), and therefore Lem. 3.6 guarantees that the heap remains well-typed when K decreases.

LEMMA 3.5. *If heap H is well-typed for K, then for every location $\mathtt{l} \in \mathrm{dom}(H)$, $\mathtt{l} \mapsto \mathtt{C<NO,NI>}(\overline{\mathtt{v}})$, then either $\mathtt{v}_i = \mathtt{null}$ or $\mathtt{l} \preceq_\theta \theta_H(\mathtt{v}_i)$.*

LEMMA 3.6. *Given a heap H that is well-typed for K, then for any $S \subset K$, the heap H is well-typed for S.*

Fig. 10 presents the reduction rules in a small-step notation, excluding all congruence rules except R-c1.

Rule R-RETURN ignores the return value of `build` and returns `l`. Rule R-FIELD-ACCESS is trivial. Rule R-FIELD-ASSIGNMENT enforces our immutability guarantee (only mutable or raw objects can be mutated) and our ownership guarantee (owner-as-dominator, i.e., `l` can point to $\mathtt{v}'$ iff $\mathtt{l} \preceq_\theta \theta_H(\mathtt{v}')$). Rule R-INVOKE finds the method body according to the receiver, and substitutes all the free variables in the method body.

Rule R-C1 is the congruence rule for `e;return l`. Note that this rule is the only place the set K is modified, i.e., when reducing `e`, the set of ongoing constructors is $K \cup \{\mathtt{l}\}$. It is easy to prove that if $K \vdash H, \mathtt{e} \to H', \mathtt{e}'$ then $\Gamma_H \subseteq \Gamma_{H'}$. The other congruence rules are not shown because they are trivial, e.g., in order to reduce a method call $\mathtt{e}_0\mathtt{.m}(\overline{\mathtt{e}})$, we first reduce $\mathtt{e}_0$ to a location, then reduce the first argument to a value, etc.

Rule R-NEW creates a new location `l`, sets the fields to `null`, sets the cooker of `l` (`VI'`) and finally calls `build`. In order to build the newly created object `l`, then it must be raw, i.e., its cooker `VI'` must be in K. (Note that `l` will be in K according to R-C1.) Therefore, if $\mathtt{VI} = \mathtt{Immut}_\mathtt{c}$ and $\mathtt{c} \notin K$, then we must set the cooker to `l`. This can happen if there is a method that returns `new C<O,I>(...)` and the receiver is a *cooked* immutable object.

3.5 Guarantees of FOIGJ

We now turn to prove various properties of FOIGJ, including preservation theorem, ownership and immutability guarantees, and an erasure property. In the remainder of this section, we assume that reduction does not get stuck on *null-pointer exceptions*, i.e., the receiver/target of field access, assignment and method calls is never `null`. Under this assumption, then `e` can always be reduced to another expression $\mathtt{e}'$.

Before stating the preservation theorem, we need to establish a connection between K and the reduced expression `e`, which may contain `return l`. Given an expression `e`, we define $K(\mathtt{e})$ to be the set of all ongoing constructors in `e`, i.e., all the locations in subexpressions `e';return l`. Formally, $K(\mathtt{e;return\ l}) = K(\mathtt{e}) \cup \{l\}$, and for any other expression we just recurse into all subexpressions, e.g., $K(\mathtt{e.f{=}e'}) = K(\mathtt{e}) \cup K(\mathtt{e}')$.

We will maintain the invariant that H is well-typed for $K \cup K(\mathtt{e})$. From Lem. 3.6, then H will also be well-typed for K. Initially, we start with a closed expression `e` without any locations (therefore $K(\mathtt{e}) = \emptyset$), an empty heap H, and an empty set of constructors K.

THEOREM 3.7. ***(Progress and Preservation)*** *For every closed expression* $\mathtt{e} \neq \mathtt{v}$*, K, and H, if* $K, \Gamma_H \vdash \mathtt{e} : \mathtt{T}$ *and H is well-typed for* $K \cup K(\mathtt{e})$*, then there exists* $H', \mathtt{e}', \mathtt{T}'$ *such that* $K \vdash H, \mathtt{e} \to H', \mathtt{e}'$*, H' is well-typed for* $K \cup K(\mathtt{e}')$*,* `T`*,* `T'`*, and* $\mathtt{e}'$ *are closed,* $K, \Gamma_{H'} \vdash \mathtt{e}' : \mathtt{T}'$*, and* $K, \Gamma_{H'} \vdash \mathtt{T}' \leq \mathtt{T}$.

Proved by showing there is always (exactly) one applicable reduction rule, which preserves subtyping. From Lem. 3.4, we know that $\mathtt{e}'$ is closed, and from Lem. 3.3, we know that `T` and `T'` are closed. Next we mention some highlights from the proof. In rule R-RETURN, we have that $K(\mathtt{e}') = K(\mathtt{e}) \setminus \{\mathtt{l}\}$, but even though we shrink K, we still have a well-typed heap from Lem. 3.6. In rule R-FIELD-ASSIGNMENT, Lem. 3.5 shows that the assumption $\mathtt{l} \preceq_\theta \theta_H(\mathtt{v}')$ holds, and the resulting heap is well-typed for $K \cup K(\mathtt{e}')$ because $K(\mathtt{e}') = \{\}$ and from T-FIELD-ASSIGNMENT. In rule R-NEW, we need to type the call $\mathtt{l.build}(\overline{\mathtt{v}})$, and for parameters with immutability `I`, we use the subtyping rule S13.

Our ownership and immutability guarantees follow directly from the reduction rules, because rule R-FIELD-ASSIGNMENT enforces them.

Thm. 3.8 shows that there is no need to maintain at run time K nor to store the owner and immutability parameter of each object. Formally, we define an erased heap structure $E(H)$ that maps location to objects without these parameters, i.e., $\mathtt{l} \mapsto \mathtt{C}(\overline{\mathtt{v}}) \in E(H)$. We define the erasure of an expression `e`, $E(\mathtt{e})$, by deleting all generic parameters, and define new reduction rules $\to_E$ in the obvious way.

THEOREM 3.8. ***(Erasure)*** *If* $K \vdash H, \mathtt{e} \to H', \mathtt{e}'$ *then*
$$K \vdash E(H), E(\mathtt{e}) \to_E E(H'), E(\mathtt{e}').$$

4. OIGJ Case Studies

This section describes our implementation of OIGJ: the language syntax (Sec. 4.1) and the type-checker implementation (Sec. 4.2). Sec. 4.3 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.

The prototype OIGJ type-checker is implemented and distributed as part of the Checker Framework [31][2], which supports pluggable type systems using type annotations.

4.1 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 [15] (e.g., `@O @I Date`). Java 7's receiver annotations play the role of cJ's guards.

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 a programmer must 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. For example, compare the annotation and generic syntax:

```
class Foo      { @O @I Bar bar; }
class Foo<O,I> { Bar<O,I>  bar; }
```

Note that the type of `new Foo<World,Immut>().bar` is explicit in the generic syntax, whereas the annotation syntax (`new @World @Immut Foo().bar`) requires additional rules that mimic generics. Use of annotations also complicates the implementation (see below). For practical use, the compati-

[2] `http://types.cs.washington.edu/checker-framework/`

bility benefits of using annotations outweigh their disadvantages.

OIGJ's annotations are the Cartesian product of owner parameters and immutability parameters. Our implementation does not yet support wildcards (though in practice the `@I` and `@O` annotations subsume most need for wildcards), nor classes with multiple owner or immutability parameters, such as `Iterator<O,ItrI,CollectionI,E>` in Fig. 3. (In our case study, we implemented iterators using a single immutability parameter by declaring `next()` as mutable.)

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

4.2 OIGJ implementation

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, as the OIGJ subtyping rule does. For example,

```
@Immut    List<@Immut    Date>   a;
@ReadOnly List<@ReadOnly Date>   b=a; // OK
@Immut    List<@Immut    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.

The OIGJ type-checker incorporates, extends, and in some places overrides the IGJ checker, and adds OGJ features. It consists of about 700 source lines of code (of which 100 lines is Java boilerplate to define the annotations). Most of the code handles default and implicit types.

4.3 `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 85 *ownership-related* annotations and 46 *immutability-related* annotations in 102 lines of code (the lines with `new` usually contain 2 annotations).

Sun's collections are not type-safe with respect to generics because Java does not support generic arrays. 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 `Raw`, thus allowing creation of immutable instances. Since all instances of `Entry` are `this`-owned, using `@This @I` as the default annotation for `Entry` meant that only three *ownership-related*[3] annotations were needed in `LinkedList`:

```
@Default({This.class, I.class})
static class Entry<E> {
  E element; @O Entry<E> next; @O Entry<E> prev;
  ... }
```

Similarly, in a `HashMap`, both the array and the entries are `this`-owned: `@This @I Entry[@This @I] table;`

The case study supports these conclusions: (i) the collections classes are properly encapsulated (they own their representation), (ii) it is possible to create immutable instances (all constructors are `Raw`), and (iii) methods `Map.get` and `clone` contain design mistakes (see below). We were not previously aware of these design mistakes. We believe that if the collections were designed 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** of Sec. 2.3, which only permits assignment to `this.header`.

```
// The following code appears in LinkedList.clone().
// Calling super.clone() breaks owner-as-dominator because
// it leaked this.header to result.header.
LinkedList result = (LinkedList) super.clone();
result.header = new Entry(); // Illegal in OIGJ!
```

We sketch a solution that, instead of initializing the cloned `result` from `this`, uses the idea of *inversion of control*. The

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

	OIGJ	OGJ [33]	IGJ [40]	GUT [14]	UTT [25]	IOJ [19]	JOE_3 [30]
Owner-as-dominator	+	+				+	+
Owner-as-modifier				+	+		
Readonly references	+		+	+	+		+
Immutable objects	+		+			+	+
Uniqueness							+
Ownership transfer					+		
Factory method pattern	+	+	+	+	+		+
Visitor pattern	+		+				
Sun's `LinkedList`	+						

Figure 11. Features supported by various language designs.

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 should be given all the privileges of a constructor, such as assignment to `final` fields.) (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 guarantees.

5. Related Work

In this section we discuss 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.

5.1 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.

Ownership Generic Java (OGJ) [33] 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 the language, the formalism, and the implementation.

OGJ completely prohibits wildcards as owner parameters, e.g., `Point<?>`, whereas OIGJ relaxes this rule and allows wildcards on stack variables, which enables writing the `equals` method (see **Generic Wildcards rule** in Sec. 2.3).

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., `O` $\preceq_\theta$ `O2`. This rule is very restrictive, however it guarantees that the ownership structure is a tree. OIGJ removed this rule at the cost of complicating the ownership structure: it is a *directed acyclic graph* (DAG) instead of a tree.

Finally, OIGJ can express temporary ownership within a method by using a fresh owner parameter (see **Fresh owners** in Sec. 2.3).

Immutability Generic Java (IGJ) [40] showed how generic types can be used to provide support for readonly references and object immutability. OIGJ used ownership information to improve the expressiveness of IGJ. Specifically, certain restrictions in IGJ no longer apply in OIGJ for `this`-owned objects. For example, `Raw` is *not* transitive in IGJ, e.g., the assignment to `next` in Fig. 3 on lines 9 and 22 is illegal in IGJ, thus limiting creation of immutable objects. In contrast, `Raw` is transitive in OIGJ for `this`-owned fields (see **Field assignment rule** in Sec. 2.3), 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. The `@Assignable` annotation indicates that a given field is not part of the object's abstract state. This is necessary to type-check caches, lazily-initialized fields, and other programming idioms. OIGJ removed this annotation to simplify the formalism. This also guarantees representation immutability as well as immutability of the abstraction: the fields of a cooked immutable object never change. Our implementation supports the `@Assignable` annotation.

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 37 in Fig. 3). In contrast, in OIGJ, we guard `next()` with a mutable `ItrI` (line 51), and guard `remove()` with a mutable `CollectionI` (line 52).

5.2 Relationship with other work

OIGJ uses method guards borrowed from *cJ* [20]. (The OIGJ implementation uses annotations syntax instead.)

In what follows, we have room to survey only closely related papers. Fig. 11 compares OIGJ to some of the previous work described below.

Mutability and encapsulation were first combined by Flexible Alias Protection (FLAP) [28]. FLAP inspired a number of proposals including ownership types [13] and confined types [36]. Capabilities for Sharing [5] describes 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* [6] 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.

Ownership types [2, 3, 11] impose a structure on the references between objects in a program's memory. OIGJ and other work [30, 33] enforce the owner-as-*dominator* disciplines. Generic Universe Types (**GUT**) [14, 24] enforce owner-as-*modifier* by using three type annotations: `rep`, `peer`, and `readonly`. `rep` denotes representation objects (similar to `This`), while `peer` denotes objects owned by the same owner (similar to `O`). **UTT** [25] is an extension of Universe Types that supports ownership transfer by utilizing a modular static analysis, which is useful for merging data-structures or complex object initialization.

MOJO [10] can express multiple ownership: objects can have more than one owner at run time. OIGJ supports only a single *owner* at run time. (OIGJ supports multiple *owner parameters*, but according to the **Ownership nesting rule**, all owner parameters are inside the first owner parameter.)

Jo∃ [8] 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 that may have different owners, from a list of student that share the same unknown owner.

Immutability and ownership. Similarly to OIGJ, Immutable Objects for a Java-like Language (**IOJ**) [19] associates with each type its mutability and owner. In contrast to OIGJ, IOJ does not have generics, nor readonly *references*. Moreover, in IOJ, the constructor cannot leak a reference to `this`. Haack and Poll [19] later added flexible initialization of immutable objects, i.e., an immutable object may still be mutated after its constructor returns. They use the annotations `RdWr`, `Rd`, `Any`, and `myaccess`, which corresponds to our `Mutable`, `Immut`, `ReadOnly`, and `I`. In addition, they have an inference algorithm that automatically infers the end of object initialization phases. (Their algorithm infers which variables are `Fresh(n)`, which resembles our `Raw`. However, the programmer cannot write the `Fresh` annotation explicitly.)

X10 [29] supports constrained types that can refer to properties and final local variables. X10 supports cyclic immutable structures by using `proto` annotations, which are similar to our immutability `I` and the notion of cookers. However, both X10 and IGJ cannot type-check Sun's `LinkedList` because an object becomes cooked when its constructor finishes. It is possible to refactor `LinkedList` to fit X10's typing-rules by using a recursive implementation, but then you risk a stack-overflow when creating large lists. Delayed types [16], which are similar to X10's `proto`, are used to verify non-null fields or other heap-monotonic properties.

JOE$_3$ [30] combines ownership (as dominators, not modifiers), uniqueness, and immutability. It also supports owner-polymorphic methods, but not existential owners.

Frozen Objects [23] show how ownership can help support immutability by allowing programmers to decide when the object should become immutable. This system takes a verification approach rather than a simple type checker such as OIGJ. Frozen Objects show how flexible the initialization stage can potentially be in the presence of ownership and immutability, while OIGJ shows how much flexibility can be achieved while staying at the type checking level.

Readonly references are found in C++ (using the `const` keyword), JAC [22], modes [34], Javari [35], etc. Previous work on readonly references lack ownership information. Boyland [4] 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 owner-as-dominator property. Both OIGJ and SafeJava [3] 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 [27].

6. Conclusion

OIGJ is a Java language extension that supports both ownership and immutability, while enhancing the expressiveness of each individual concept. By using Java's generic types, OIGJ simplifies previous type mechanisms, such as existential owners, scoped regions, and owner-polymorphic methods. OIGJ is easy to understand and implement, using only 14 (flow-insensitive) typing rules beyond those of Java. We have formalized a core calculus called FOIGJ and proved it sound. Our implementation is backward-compatible with Java, and it scales to realistic programs. OIGJ can type-check Sun's `java.util` collections (without the `clone` method), using a small number of annotations. Finally, various design patterns, such as the factory and visitor patterns, can be expressed in OIGJ, making it ready for practical use. An implementation is publicly available at `http://types.cs.washington.edu/checker-framework/`.

Future work includes inferring ownership and immutability, conducting a bigger case study with client and library code, and extending OIGJ with concepts such as *owner-as-modifier* [41], *uniqueness*, and *external-uniqueness* [12].

Acknowledgments

This work was supported by the grants from RSNZ Marsden, ISAT, and NSF CNS-0855252. We especially thank Werner Dietl, James Noble, ELVIS, and the anonymous referees.

References

[1] Chandrasekhar Boyapati. *SafeJava: A Unified Type System for Safe Programming*. PhD thesis, MIT Dept. of EECS, Feb. 2004.

[2] Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *OOPSLA*, pages 211–230, Oct. 2002.

[3] Chandrasekhar Boyapati, Barbara Liskov, and Liuba Shrira. Ownership types for object encapsulation. In *POPL*, pages 213–223, Jan. 2003.

[4] John Boyland. Why we should not add `readonly` to Java (yet). In *FTfJP*, July 2005.

[5] John Boyland, James Noble, and William Retert. Capabilities for sharing: A generalisation of uniqueness and read-only. In *ECOOP*, pages 2–27, June 2001.

[6] John Boyland, William Retert, and Yang Zhao. Comprehending annotations on object-oriented programs using fractional permissions. In *IWACO*, pages 1–11, July 2009.

[7] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In *OOPSLA*, pages 183–200, Oct. 1998.

[8] Nicholas Cameron and Sophia Drossopoulou. Existential quantification for variant ownership. In *ESOP*, pages 128–142, Mar. 2009.

[9] Nicholas Cameron, Sophia Drossopoulou, and Erik Ernst. A model for Java with wildcards. In *ECOOP*, pages 2–26, July 2008.

[10] Nicholas R. Cameron, Sophia Drossopoulou, James Noble, and Matthew J. Smith. Multiple ownership. In *OOPSLA*, pages 441–460, Oct. 2007.

[11] Dave Clarke and Sophia Drossopoulou. Ownership, encapsulation and the disjointness of type and effect. In *OOPSLA*, pages 292–310, Oct. 2002.

[12] Dave Clarke and Tobias Wrigstad. External uniqueness is unique enough. In *ECOOP*, pages 176–200, July 2003.

[13] David G. Clarke, John M. Potter, and James Noble. Ownership types for flexible alias protection. In *OOPSLA*, pages 48–64, Oct. 1998.

[14] Werner Dietl, Sophia Drossopoulou, and Peter Müller. Generic Universe Types. In *ECOOP*, pages 28–53, Aug. 2007.

[15] Michael D. Ernst. Type Annotations specification (JSR 308). `http://types.cs.washington.edu/jsr308/`, Sep. 12, 2008.

[16] Manuel Fähndrich and Songtao Xia. Establishing object invariants with delayed types. In *OOPSLA*, pages 337–350, Oct. 2007.

[17] Erich Gamma, Richard Helm, Ralph E. Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley, Reading, MA, 1995.

[18] Christian Haack and Erik Poll. Type-based object immutability with flexible initialization. In *ECOOP*, pages 520–545, 2009.

[19] Christian Haack, Erik Poll, Jan Schäfer, and Aleksy Schubert. Immutable objects for a Java-like language. In *ESOP*, pages 347–362, Mar. 2007.

[20] Shan Shan Huang, David Zook, and Yannis Smaragdakis. cJ: Enhancing Java with safe type conditions. In *AOSD*, pages 185–198, Mar. 2007.

[21] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM TOPLAS*, 23(3):396–450, May 2001. ISSN 0164-0925.

[22] Günter Kniesel and Dirk Theisen. JAC — access right based encapsulation for Java. *Software: Practice and Experience*, 31(6):555–576, 2001.

[23] K. Rustan M. Leino, Peter Müller, and Angela Wallenburg. Flexible immutability with frozen objects. In *VSTTE*, pages 192–208, Oct. 2008.

[24] P. Müller and A. Poetzsch-Heffter. Universes: A type system for controlling representation exposure. In *Programming Languages and Fundamentals of Programming*, pages 131–140, 1999.

[25] Peter Müller and Arsenii Rudich. Ownership transfer in universe types. In *OOPSLA*, pages 461–478, Oct. 2007.

[26] Stefan Nägeli. Ownership in design patterns. Master's thesis, ETH Zürich, Zürich, Switzerland, Mar. 2006.

[27] James Noble. Iterators and encapsulation. In *TOOLS Pacific*, pages 431–442, 2000.

[28] James Noble, Jan Vitek, and John Potter. Flexible alias protection. In *ECOOP*, pages 158–185, July 1998.

[29] Nathaniel Nystrom, Vijay Saraswat, Jens Palsberg, and Christian Grothoff. Constrained types for object-oriented languages. In *OOPSLA*, pages 457–474, Oct. 2008.

[30] Johan Östlund, Tobias Wrigstad, and Dave Clarke. Ownership, uniqueness and immutability. In *Tools Europe*, pages 178–197, 2008.

[31] Matthew M. Papi, Mahmood Ali, Telmo Luis Correa Jr., Jeff H. Perkins, and Michael D. Ernst. Practical pluggable types for Java. In *ISSTA*, pages 201–212, July 2008.

[32] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[33] Alex Potanin, James Noble, Dave Clarke, and Robert Biddle. Generic Ownership for Generic Java. In *OOPSLA*, pages 311–324, Oct. 2006.

[34] Mats Skoglund and Tobias Wrigstad. A mode system for read-only references in Java. In *FTfJP*, June 2001.

[35] Matthew S. Tschantz and Michael D. Ernst. Javari: Adding reference immutability to Java. In *OOPSLA*, pages 211–230, Oct. 2005.

[36] Jan Vitek and Boris Bokowski. Confined types. In *OOPSLA*, pages 82–96, Nov. 1999.

[37] Tobias Wrigstad. *Ownership-Based Alias Management*. PhD thesis, Royal Institute of Technology, Sweden, May 2006.

[38] Tobias Wrigstad and Dave Clarke. Existential owners for ownership types. *J. Object Tech.*, 6(4):141–159, May–June 2007.

[39] Yoav Zibin. Featherweight Ownership and Immutability Generic Java (FOIGJ). Technical Report 10-16, ECS, VUW, June 2010. `http://ecs.victoria.ac.nz/Main/TechnicalReportSeries`.

[40] Yoav Zibin, Alex Potanin, Mahmood Ali, Shay Artzi, Adam Kieżun, and Michael D. Ernst. Object and reference immutability using Java generics. In *ESEC/FSE*, pages 75–84, Sep. 2007.

[41] Yoav Zibin, Alex Potanin, Paley Li, Mahmood Ali, and Michael D. Ernst. OIGJ with owners as modifiers. Technical Report 10-15, ECS, VUW, January 2010.