Computer Science and Artificial Intelligence Laboratory
Technical Report

MIT-CSAIL-TR-2007-018 March 16, 2007

Object and Reference Immutability using Java Generics

Yoav Zibin, Alex Potanin, Shay Artzi , Adam Kiezun and Michael D. Ernst

massachusetts institute of technology, cambridge, ma 02139 usa — www.csail.mit.edu

Object and Reference Immutability using Java Generics

Yoav Zibin
MIT Computer Science and Artificial Intelligence Lab
zyoav@csail.mit.edu

Alex Potanin
Victoria University of Wellington
alex@mcs.vuw.ac.nz

Shay Artzi Adam Kieżun
Michael D. Ernst
MIT Computer Science and Artificial Intelligence Lab
{artzi | akiezun | mernst}@csail.mit.edu

Abstract

A compiler-checked immutability guarantee provides useful documentation, facilitates reasoning, and enables optimizations. This paper presents *Immutability Generic Java* (IGJ), a novel language extension that expresses immutability without changing Java's syntax by building upon Java's generics and annotation mechanisms. In IGJ, each class has one additional generic parameter that is `Immutable`, `Mutable`, or `ReadOnly`. IGJ guarantees both *reference immutability* (only mutable references can mutate an object) and *object immutability* (an immutable reference points to an immutable object). IGJ is the first proposal for enforcing object immutability, and its reference immutability is more expressive than previous work. IGJ also permits covariant changes of generic arguments in a type-safe manner, e.g., a readonly list of integers is a subtype of a readonly list of numbers. IGJ extends Java's type system with a few simple rules. We formalize this type system and prove it sound. Our IGJ compiler works by type-erasure and generates byte-code that can be executed on any JVM without runtime penalty.

1. Introduction

Immutability information is useful in many software engineering tasks, such as modeling [7], verification [31], compile- and run-time optimizations [10, 25, 29], refactoring [17], test input generation [1], regression oracle creation [24, 33], invariant detection [14], specification mining [11], and program comprehension [21]. Three varieties of immutability guarantee are:

Class immutability No instance of an *immutable class* may be changed; examples in Java include `String` and most subclasses of `Number` such as `Integer` and `BigDecimal`.

Object immutability An *immutable object* can not be modified, even if other instances of the same class can be. For example, some instances of `List` in a given program may be immutable, whereas others can be modified. Object immutability can be used for pointer analysis and optimizations, such as sharing between threads without synchronization, and to help prevent hard-to-detect bugs, e.g., the documentation of the `Map` interface in Java states that "Great care must be exercised if *mutable objects* are used as map keys. The behavior of a map is not specified if the value of an object is changed in a manner that affects equals comparisons while the object is a key in the map."

Reference immutability A *readonly reference* [2, 5, 13, 22, 25, 30, 32] (or a *const* pointer in C++) cannot be used to modify its referent. (However, the referent might be modified using an aliasing mutable reference.) Reference immutability is required to specify interfaces, such as that a procedure may not modify its arguments (even if the caller retains the right to do so) or a client may not modify values returned from a module. However, past work does not guarantee object immutability, unless reference immutability is combined with an alias/escape analysis to guarantee that no aliases to an object exist.

This paper presents *Immutability Generic Java* (IGJ), a language that supports class, object, and reference immutability. Each object is either mutable or immutable, and each reference is `Immutable`, `Mutable`, or `ReadOnly`. Inspired by work that combines ownership and generics [27], the distinctions are expressed without changing Java's syntax by adding one new generic parameter (at the beginning of the list of generic parameters):

```
1: // An immutable reference to an immutable date; mutating the
   referent is prohibited, via this or any other reference.
2: Date<Immutable> immutD = new Date<Immutable>();
3: // A mutable reference to a mutable date; mutating the referent
   is permitted, via this or any other mutable reference.
4: Date<Mutable> mutD = new Date<Mutable>();
5: // A readonly reference to any date; mutating the referent is
   prohibited via this reference, but the referent may be changed
   via an aliasing mutable reference.
6: Date<ReadOnly> roD = ... ? immutD : mutD;
```

Line 2 shows object immutability in IGJ, and line 6 shows reference immutability. See Fig. 4 for a larger example of IGJ code.

Java prohibits changes to generic arguments, such as in line 6, to avoid a type loophole. However, if mutation is disallowed, then such covariant changes are type-safe, and indeed in IGJ `List<ReadOnly,Integer>` is a subtype of `List<ReadOnly,Number>`.

IGJ satisfies the following design principles:

Transitivity IGJ provides transitive (deep) immutability that protects the entire abstract state of an object. For example, an immutable graph can contain an immutable set of immutable edges. C++, for example, does not support such transitivity because C++'s `const`-guarantee does not traverse pointers, i.e., a pointer in a `const` object can mutate the object to which it points.

Moreover, IGJ supports *shallow immutability*, i.e., it is possible to exclude some fields from the abstract state. For example, fields used for caching can be mutated even in an immutable object.

Static There is no runtime representation for immutability, such as an "immutability bit" that is checked before assignments or method calls. Testing at runtime whether an object is immutable [30] hampers program understanding.

The IGJ compiler works by type-erasure, without any run-time representation of reference or object immutability, which enables executing the resulting code on any JVM without run-

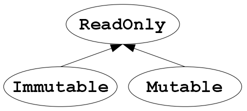

Figure 1. The type hierarchy of immutability parameters, which have special meaning when used as the first generic parameter, as in `List<Mutable,T>`. (See also Fig. 6 in Sec. 2.4)

time penalty. A similar approach was taken by Generic Java (GJ) [6] that extended Java 1.4. As with GJ, libraries must either be retrofitted with IGJ types, or fully converted to IGJ, before clients can be compiled. IGJ is backward compatible, i.e., every legal Java program is a legal IGJ program.

Polymorphism It is possible to abstract over immutability without code duplication, e.g., a method argument that is either mutable or immutable. For instance, all the collection classes in C++'s STL have two overloaded versions of `iterator`, `operator[]`, etc. The underlying problem is the inability to return a reference whose immutability depends on the immutability of `this`:

```
const Foo& getFieldFoo() const;
      Foo& getFieldFoo();
```

Simplicity IGJ does not change Java's syntax or typing rules. IGJ adds a small number of additional typing rules, that make IGJ more restrictive than Java. On the other hand, IGJ subtyping rules are more relaxed, allowing covariant changes in a type-safe manner.

The contributions of this paper are as follows: (i) a novel and simple design combining both reference and object immutability that naturally fits into Java's generics framework, (ii) an implementation of an IGJ compiler, proving feasibility of the design, and (iii) a formalization of IGJ with a proof of type soundness. Our ideas, though demonstrated using Java, are applicable to any statically typed language with generics, such as C++, C#, and Eiffel.

Outline The remainder of this paper is organized as follows. Sec. 2 describes the IGJ language, which is compared to previous work in Sec. 3. Sec. 4 discusses our experimentation with IGJ, and Sec. 5 formalizes IGJ and gives a proof of soundness. Sec. 6 outlines future work. We conclude in Sec. 7.

2. IGJ language

IGJ provides three kinds of references: readonly, mutable, and immutable. The first generic argument of a class/type in IGJ is called the *immutability parameter*, and it must be `Mutable`, `Immutable`, or `ReadOnly`.

2.1 Type hierarchy

Fig. 1 depicts the type hierarchy of immutability parameters. The subtyping relation is denoted by $\preceq$, e.g., `Mutable` $\preceq$ `ReadOnly`. The immutability parameters `Mutable`, `Immutable`, and `ReadOnly` may not be extended, and they have no subtype relation with any other types in IGJ.

The root of the IGJ type hierarchy (excluding `ReadOnly` and its descendants) is `Object<ReadOnly>`. Fig. 2 depicts the subtype relation for the classes `Object` and `Date`.

In Java, all generic parameters are invariant. The subtyping rules for IGJ are more relaxed. IGJ permits covariant changes in the immutability parameter (the first generic argument), e.g.,

`Date<Mutable>` $\preceq$ `Date<ReadOnly>`

This satisfies the polymorphism design principle, because a programmer can write a single method that accepts a reference of any immutability, for example:

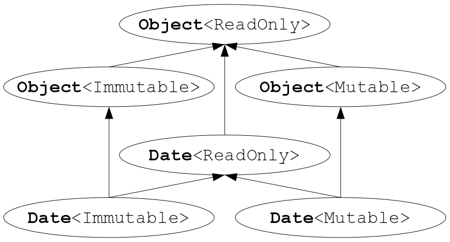

Figure 2. The subtype hierarchy for `Object` and `Date`. The classes (in bold) still have an underlying tree structure.

```
void print(Date<ReadOnly> d)
```

IGJ also permits covariant changes in other generic arguments if the immutability parameter is `ReadOnly` or `Immutable`, e.g.,

`List<ReadOnly,Integer>` $\preceq$ `List<ReadOnly,Number>`

Covariant changes are safe only when the object is readonly or immutable because it cannot be mutated in a way which is not type-safe. Therefore the following pair is not in the subtype relation:

`List<Mutable,Integer>` $\not\preceq$ `List<Mutable,Number>`

A type variable `X` in class `C` can be annotated with `@NoVariant` to prevent covariant changes, in which case we say that `X` is no-variant and write *NoVariant*$(\texttt{X},\texttt{C})$. Otherwise we say that `X` is co-variant and write *CoVariant*$(\texttt{X},\texttt{C})$. Sections 2.3 and 2.5 discuss when a type-variable should be no-variant. For example, the type variable `X` in the interface `Comparable<I,X>` is no-variant:[1]

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

For instance, `Comparable<ReadOnly,Integer>` was written to compare itself to an `Integer`, thus it should not be a subtype of `Comparable<ReadOnly,Number>`, that can be passed a `Number`.

IGJ's subtype definition for two types of the same class is given in Def. 2.1. The full subtype definition (formally given in Fig. 12 of Sec. 5) includes all of Java's subtyping rules, therefore IGJ's subtype relation is a superset of Java's subtype relation.

Definition 2.1. *Let* $\texttt{C<I,X}_1,\ldots,\texttt{X}_n\texttt{>}$ *be a class with* $n \geq 0$ *generic variables. Then, type* $\texttt{S} = \texttt{C<J,S}_1,\ldots,\texttt{S}_n\texttt{>}$ *is* a subtype of $\texttt{T} = \texttt{C<J}',\texttt{T}_1,\ldots,\texttt{T}_n\texttt{>}$, *written as* $\texttt{S} \preceq \texttt{T}$, *iff* $\texttt{J} \preceq \texttt{J}'$ *and for* $i = 1,\ldots,n$, *either* $\texttt{S}_i = \texttt{T}_i$ *or* ($\texttt{Immutable} \preceq \texttt{J}'$ *and* $\texttt{S}_i \preceq \texttt{T}_i$ *and* *CoVariant*$(\texttt{X}_i,\texttt{C})$).

Example Fig. 3 presents the subtype hierarchy for `List<Object>`. The types `L<M,O<M>>`, `L<M,O<IM>>`, and `L<M,O<R>>` have a common *mutable* supertype `L<M,? extends O<? extends R>>`, but the only value that can be inserted in such a list is `null`. (See Sec. 3 for a discussion of Java's wildcards.)

2.2 Reference immutability

This section gives the three key type rules of IGJ that enforce reference immutability: that is, only a `Mutable` reference can modify its referent. To support reference immutability it is sufficient to use `ReadOnly` and `Mutable` references; Sec. 2.4 adds object immutability by using `Immutable` references as well.

We use $I(\ldots)$ to denote a function that takes a class, type, or reference, and returns its immutability (parameter). For example, given a reference `mutD` of type `Date<Mutable>`, $I(\texttt{mutD}) = \texttt{Mutable}$, and we say that the *immutability of* `mutD` is `Mutable`.

A field can be assigned only by a mutable reference:

[1] Annotating generic arguments will be available only in Java 7 [15]. In the meantime it is possible to annotate the class or interface instead, and specify which positions are no-variant.

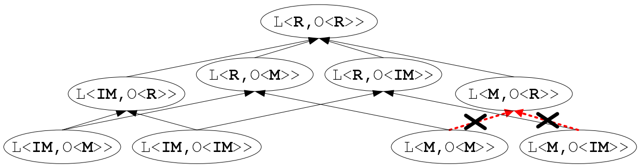

Figure 3. The subtype hierarchy for `List<Object>`, abbreviated as `L<O>`. The types `ReadOnly`, `Mutable`, and `Immutable` are abbreviated as `R`, `M`, and `IM`, respectively. The crossed-out arrows emphasize pairs that are *not* subtypes.

FIELD-ASSIGNMENT RULE:
`o.someField = ...` is legal iff $I(\texttt{o}) = \texttt{Mutable}$.

Phrased differently, you cannot assign to fields of a readonly reference, e.g.,

```
Employee<ReadOnly> roE = ...;
roE.address = ...; // Compilation error!
```

The immutability of `this` depends on the context, i.e., the method in which `this` appears:

THIS RULE: $I(\texttt{this}) = I(\texttt{m})$, in a method `m`.

Because there is no obvious syntactic way to denote the immutability of `this`, IGJ uses method annotations: `@ReadOnly`, `@Mutable`, etc.[2]

For example, below we have $I(\texttt{this}) = I(\texttt{m}) = \texttt{Mutable}$:

```
@Mutable void m() { ... this ...}
```

The default method annotation in IGJ is `@Mutable`, but for clarity of presentation, this paper explicitly annotates all methods.

The third type rule of IGJ states when a method call is legal:

METHOD-INVOCATION RULE: `o.m(...)` is legal iff $I(\texttt{o}) \preceq I(\texttt{m})$.

IGJ requires that $I(\texttt{o}) \preceq I(\texttt{m})$, and not simply $I(\texttt{o}) = I(\texttt{m})$, to allow a mutable reference to call readonly methods, e.g.,

```
Employee<Mutable> o = ...;
o.setAddress(...); // OK: I(o) ⪯ I(setAddress) = Mutable
o.getAddress();    // OK: I(o) ⪯ I(getAddress) = ReadOnly
((Employee<ReadOnly>) o).setAddress(...); // Illegal!
```

Example Fig. 4 presents two IGJ classes: `Edge` and `Graph`. The immutability parameter `I` is declared in lines 1 and 11; by convention we always denote it by `I`. If the `extends` clause is missing from a class declaration, then we assume that class extends `Object<I>`. We can use any subtype of `ReadOnly` in place of `I`, e.g., `ReadOnly` (on line 9), `Mutable` (on line 15), or another parameter such as `I` (on line 12) or `X` (on line 18).

We will now demonstrate the type-checking rules by example. The assignment `this.id = id` on line 5 is legal because according to THIS RULE we have that $I(\texttt{this}) = I(\texttt{setId}) = \texttt{Mutable}$, and according to FIELD-ASSIGNMENT RULE a mutable reference can assign to a field. That assignment would be illegal if it was moved to line 6, because `this` is readonly in the context of method `getId`. The method call `this.setId(...)` on line 3 is legal according to METHOD-INVOCATION RULE because $I(\texttt{this}) \preceq I(\texttt{setId})$. That method call would be illegal on line 6.

Observe on line 9 that the static method `print` does not have an annotation because it does not have an associated `this` object. According to Def. 2.1 of the subtype relation, an edge of any immutability can be passed to `print`.

```
 1: class Edge<I extends ReadOnly> {
 2:   private long id;
 3:   @AssignsFields Edge(long id) { this.setId(id); }
 4:   @AssignsFields synchronized void setId(long id) {
 5:       this.id = id; }
 6:   @ReadOnly synchronized long getId() { return id; }
 7:   @Immutable long getIdImmutable() { return id; }
 8:   @ReadOnly Edge<I> copy() { return new Edge<I>(id); }
 9:   static void print(Edge<ReadOnly> e) { ...}
10: }
11: class Graph<I extends ReadOnly> {
12:   List<I,Edge<I>> edges;
13:   @AssignsFields Graph(List<I,Edge<I>> edges) {
14:       this.edges = edges; }
15:   @Mutable void addEdge(Edge<Mutable> e) {
16:       this.edges.add(e); }
17:   static <X extends ReadOnly> Edge<X>
18:       findEdge(Graph<X> g, long id) { ...}
19: }
```

Figure 4. IGJ classes `Edge<I>` and `Graph<I>`, with the immutability parameters (and annotations, for `this`) underlined. Erasing the immutability parameters and annotations yields a legal Java program with the same semantics. The annotations `@Immutable` and `@AssignsFields` are explained in Sec. 2.4; for now assume that `@Immutable` is the same as `@ReadOnly`, and `@AssignsFields` is the same as `@Mutable`.

We call the field `edges` on line 12 `this`-*mutable* [32] because its immutability depends on the immutability of `this`: in a mutable object this field is mutable and in a readonly object it is readonly. C++ has similar behavior for fields without the keywords `const` or `mutable`. The advantage of IGJ syntax is that the concept of `this`-*mutable* is made explicit in the syntax: a class can pass its immutability parameter to its fields, and the underlying generic type system propagates the immutability information without the need for special type-rules. Using generics simplifies both the design and the implementation.

Moreover, C++ has no `this`-mutable local variables, return types, or parameters, whereas IGJ treats `I` as a regular generic parameter. For example, the parameter `edges` on line 13 and the return type on line 8 are both `this`-mutable.

Recall that the **Transitivity** design principle states that the design must support transitive (deep) immutability. In our example, in a mutable `Graph` the field `edges` will contain a mutable list of *mutable* edges. The second usage of `I` in `List<I,Edge<I>>` is not expressible in C++.

2.3 Method overriding

IGJ respects the Java class hierarchy. An overriding method must preserve or strengthen the specification of the overridden method:

METHOD-OVERRIDING RULE 1:
If method `m'` overrides `m`, then $I(\texttt{m}) \preceq I(\texttt{m'})$.

For example, overriding can change a `@Mutable` method to a `@ReadOnly` method, but not vice versa.

IGJ requires that the *erased signature* of an overriding method remain the same if that method is either readonly or immutable. The *erased signature* of a method is obtained by replacing type-variables with their bounds. When the erased signature of an overriding method changes, the compiler inserts a *bridge method* to cast the arguments to the correct type [6].

METHOD-OVERRIDING RULE 2:
If method `m'` overrides `m` and $\texttt{Immutable} \preceq I(\texttt{m})$, then the erased signatures of `m'` and `m` (excluding no-variant type-variables) must be identical.

[2] The paper uses the annotation `@ReadOnly` whereas the IGJ compiler uses `@ReadOnlyMethod`, because an annotation and a class cannot have the same qualified name. The same applies for the other three annotations.

```
1: class MyVector<I extends ReadOnly, X> { ...
2:   @ReadOnly void isIn(X o) {...} // The erased signature is isIn(Object)
3: }
4: class MyIntVector<I extends ReadOnly> extends MyVector<I,Integer> { ...
5:   // Overriding isIn is illegal due to METHOD-OVERRIDING RULE 2: the erased signature isIn(Integer) is different from isIn(Object)
6:   @ReadOnly void isIn(Integer o) {...} // Would be legal if X was annotated with @NoVariant
7: }
8: MyVector<ReadOnly,Object> v = new MyIntVector<ReadOnly>(); // Would be illegal if X was annotated with @NoVariant
9: v.isIn( new Object() ); // If overriding were legal, the bridge method of isIn(Integer) would cast an Object to an Integer
```

Figure 5. An example of illegal method-overriding due to METHOD-OVERRIDING RULE 2

Fig. 5 demonstrates why METHOD-OVERRIDING RULE 2 prohibits method overriding if the erased signature changes. As another example, if `X` was annotated as `@NoVariant` in line 2, then the overriding in line 6 would be legal, and covariantly changing `X` in line 9 would be illegal.

Out of 82,262 methods in Java SDK 1.5, 30,169 methods override other methods, out of which only 51 have a different erased signature, and only the method `compareTo(X)` is readonly (the rest are mutable: `add`, `put`, `offer`, `create`, and `setValue`). Because `X` is no-variant in the `Comparable` interface, we conclude that METHOD-OVERRIDING RULE 2 does not impose any restrictions on Java SDK.

2.4 Object immutability

One advantage of *object* immutability is enabling safe sharing between different threads without the cost of synchronization. Consider lines 6–7 in Fig. 4. A `long` read/write is not atomic in Java; synchronization is necessary. However, only an immutable `Edge` can use `getIdImmutable()` to avoid the cost of synchronization.

The referent of a readonly reference (Sec. 2.2) is not immutable: it could be changed via another pointer. A separate analysis can indicate some cases when such changes are impossible [32], but it is preferable for the type system to guarantee that the referent of immutable references cannot change.

The IGJ type system makes such a guarantee:

A mutable reference points to a mutable object, and
an immutable reference points to an immutable object. (1)

In order to enforce this property, no immutable reference may be aliased by a mutable one; equivalently, no mutable reference may point to an immutable object.

2.4.1 Constructors and Annotation `@AssignsFields`

The rules given so far are sufficient to guarantee object immutability for IGJ with the exception of constructors. A constructor that is making an immutable object must be able to set fields in the object (which becomes immutable as soon as the constructor returns). It is not acceptable to mark the constructor of an immutable object as `@Mutable`, which would permit arbitrary side effects, possibly including making mutable aliases to `this`.

IGJ uses a fourth kind of reference immutability, `AssignsFields`, to permit constructors to perform limited side effects without permitting modification of immutable objects. Whereas a `@Mutable` method can assign *and mutate* the object's fields, an `@AssignsFields` method can only assign (*not mutate*) the fields of <u>`this`</u>. A programmer can write the `@AssignsFields` method annotation but may *not* write the `AssignsFields` type in any other way, such as `Edge<AssignsFields>`. Therefore, in an `@AssignsFields` constructor, `this` can only escape as `ReadOnly`. Fig. 6 shows the full hierarchy of immutability parameters.

`AssignsFields` is not transitive, i.e., you can assign to fields of `this` but not to fields of fields of `this`. Specifically, we relax FIELD-ASSIGNMENT RULE by allowing a field of `this` to be assigned in an `@AssignsFields` method:

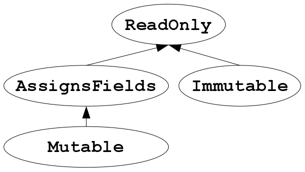

Figure 6. The type hierarchy of immutability parameters, including `AssignsFields`.

FIELD-ASSIGNMENT RULE revised:
`o.someField = ...` is legal iff
$I(\texttt{o}) = \texttt{Mutable}$ or ($I(\texttt{o}) = \texttt{AssignsFields}$ and $\texttt{o} = \underline{\texttt{this}}$).

Next, we restrict the METHOD-INVOCATION RULE:

METHOD-INVOCATION RULE revised:
`o.m(...)` is legal iff
$I(\texttt{o}) \preceq I(m)$ and ($I(\texttt{m}) = \texttt{AssignsFields}$ implies $\texttt{o} = \underline{\texttt{this}}$).

(The addition of $\texttt{o} = \underline{\texttt{this}}$ ensures that `AssignsFields` is not transitive.)

It is not always known at compile time whether a `new` operation creates a mutable or an immutable object, e.g., line 8 of Fig. 4. IGJ always permits creating an object using a `@ReadOnly` or `@AssignsFields` constructor, however it permits using a `@Mutable` constructor only for creating a mutable object (and similarly for an `@Immutable` constructor):

OBJECT-CREATION RULE: `new SomeClass<X,...>(...)` is *illegal* iff the annotation `Y` of the constructor satisfies:
$\texttt{Y} = \texttt{@Mutable}$ and $\texttt{X} \neq \texttt{Mutable}$, or
$\texttt{Y} = \texttt{@Immutable}$ and $\texttt{X} \neq \texttt{Immutable}$.

Example The assignment `this.id = ...`, on line 5 of Fig. 4, is legal according to the new FIELD-ASSIGNMENT RULE because method `setId` is annotated with `@AssignsFields` and thus the immutability of `this` is $I(\texttt{this}) = \texttt{AssignsFields}$. The method call `this.setId(...)` on line 3 was already legal (and is still legal) because

$I(\texttt{this}) = \texttt{AssignsFields} \preceq I(\texttt{setId}) = \texttt{AssignsFields}$.

The METHOD-INVOCATION RULE was made more strict to avoid transitivity. For instance, adding the code `this.edges.get(0).setId(42)` to line 14 is legal in the old METHOD-INVOCATION RULE but not in the revised one. Note that this addition must be illegal because it could mutate an immutable edge in the list `edges`.

A field of an immutable object can be assigned multiple times in the constructor or even in other `@AssignsFields` methods. This is harmless, and the programmer can mark a field as `final` to ensure that it is assigned in the constructor once and no more than once.

Field initializers Field initializers are expressions that are used to initialize the object's fields. It is safe to consider the immutability of `this` as `AssignsFields` in such expressions. However, if all

```
1: class AccessOrderedSet<I extends ReadOnly,
2:     @NoVariant X> {
3:   private List<Mutable, X> l;
4:   public @ReadOnly boolean contains(X x) {
5:     ...
6:     // We can mutate this.l even though this is ReadOnly
7:     this.l.addFirst(x);
8:   }
9: }
```

Figure 7. Class `AccessOrderedSet` with a mutable field `l`. Variable `X` is no-variant.

constructors are mutable, we can safely assume that `this` is mutable as well.

Remark 2.2. *Before JSR 133 [28] immutable objects could have different values in different threads. The new memory model adds a "synchronization point" at the end of a constructor after which all assignments are guaranteed to be written. However, the programmer is warned (in the documentation only!) that immutable objects are thread-safe only if* `this` *does not escape its constructor. Therefore we plan to modify IGJ to issue a warning whenever* `this` *escapes, even though it is not dangerous to IGJ's type system because only a readonly* `this` *can escape.*

2.5 Mutable and assignable fields

A type system should guarantee facts about the abstract state of an object, not merely its concrete representation. Therefore, a transitive guarantee of immutability for *all* fields of an object may be too strong. For example, fields used for caching are not part of the abstract state. This section discusses how to permit a given field to be assigned or mutated even in a readonly or immutable object, and than discusses special restrictions involving such fields.

Assignable fields An assignable field (denoted by the `@assignable` annotation) is in essence the reverse of a final field: a final field cannot be re-assigned whereas an assignable field can always be assigned (even using a readonly reference). IGJ uses to denote that such a field can always be assigned. We revise FIELD-ASSIGNMENT RULE to always allow assigning to an assignable field:

> **FIELD-ASSIGNMENT RULE revised again**:
> `o.someField = ...` is legal iff
> $I(\texttt{o}) = \texttt{Mutable}$ or ($I(\texttt{o}) = \texttt{AssignsFields}$ and $\texttt{o} = \underline{\texttt{this}}$) or `someField` is annotated as `@assignable`.

For example, consider this code snippet:

```
private @assignable int memoizedHashCode = 0;
public @ReadOnly int hashCode() {
  if (memoizedHashCode == 0) {
    memoizedHashCode = ...;
  }
  return memoizedHashCode;
}
```

The assignment `memoizedHashCode=...` is legal even though `hashCode` is readonly, due to the `@assignable` annotation.

Mutable fields A *mutable field* can always be mutated, even using a readonly reference. No new linguistic mechanism is required to express a mutable field: its immutability parameter is `Mutable`.

For instance, `AccessOrderedSet` in Fig. 7 implements a set using a list `l` (line 3), and as an optimization it maintains the list in access-order even during calls to readonly methods such as `contains` (line 7). Therefore, `l` is declared as a mutable field.

Covariant and no-variant type-variables Type-variable `X` in Fig. 7 must be annotated as `@NoVariant` (line 2) due to its use in the mutable field `l`. If `X` could change covariantly, we would have that:

```
AccessOrderedSet<ReadOnly, Integer > ⪯
AccessOrderedSet<ReadOnly, Number  >
```

We could than add a `Number` to an `Integer` list using the `contains` method in line 3 of Fig. 7.

An assignable field or a mutable superclass have the same restriction as a mutable field:

> **NOVARIANT RULE**: A type-variable must be no-variant if it is used in a mutable field, an assignable field, a mutable superclass, or in the position of another no-variant type-variable. (See a formal definition in Fig. 11.)

The immutability parameter is a special generic variable that must be allowed to change covariantly, otherwise a mutable reference could not call a readonly method. In other words, `I` must be co-variant:

> **COVARIANT RULE**: *CoVariant*$(\texttt{I}, \texttt{C})$ must hold for any class `C`.

For example, declaring the following field is prohibited because the immutability parameter `I` must always be co-variant:

```
@assignable Edge<I> f;
```

2.6 Inner classes

Nested classes that are *static* can be treated the same as normal classes. An *inner class* is a nested class that is not explicitly or implicitly declared static (see JLS 8.1.3 [18]). Inner classes have an additional `this` reference: `OuterClass.this`. According to THIS RULE, the immutability of `this` depends on the immutability of the method. Because methods in IGJ have a single method annotation, the immutability of `this` and `OuterClass.this` should be the same. Therefore, in IGJ an inner class cannot have its own immutability parameter:

> **INNER-CLASS RULE**: An inner class *inherits the immutability parameter* of the outer class.

Consider the code in Fig. 8. Lines 1–5 show the declaration of the `Iterator` interface, in which the only mutable method is `remove`. The immutability of an *iterator* is inherited from its *container*. Therefore, even though method `next` changes the state of the *iterator* it does not change the state of the *container* and is thus considered state preserving for the container and declared as readonly. In contrast, method `remove` changes the container and is thus marked as mutable.

Now consider the class `ArrayList` and the inner class `ArrItr`. We do not pass an immutability parameter to the inner class `ArrItr` on lines 7–8, because it inherits its immutability from `ArrayList`. On line 13 both `this` references are mutable because `remove()` is mutable. Finally, consider the creation of a new iterator on line 7. We handle this new operation using METHOD-INVOCATION RULE for method calls (instead of OBJECT-CREATION RULE): this method call is legal because `this` is readonly and the constructor of `ArrItr` is readonly. We do not use OBJECT-CREATION RULE because the inner object inherits the immutability of the outer object.

Anonymous inner classes can be considered identical to regular inner classes where the only exception is that we cannot write a constructor because the class has no name, making it impossible to annotate the constructor. For instance, the code in Fig. 8 can be converted to use an anonymous inner class:

```
public @ReadOnly Iterator<I,E> iterator() {
  return new Iterator<I,E>() { ... };
}
```

However, we can safely assume that the immutability of the missing constructor is the same as the immutability of the method in which the inner class is declared, or else the new operation would

```
interface Iterator<I extends ReadOnly,E> {
  @ReadOnly boolean hasNext();
  @ReadOnly E next();
  @Mutable void remove();
}
class ArrayList<I extends ReadOnly, E> ...   { ...
  public @ReadOnly Iterator<I,E> iterator() { return this.new ArrItr(); } // OK: I(this) ⪯ I(ArrItr) = ReadOnly
  class ArrItr implements Iterator<I,E> { // ArrItr has no explicit immutability parameter: it is inherited from the outer class
    private @assignable int currPos;
    public @ReadOnly ArrItr() { this.currPos=0; } // OK: currPos is @assignable
    public @ReadOnly boolean hasNext() { return this.currPos < ArrayList.this.size(); }
    public @ReadOnly E next() { return ArrayList.this.get( this.currPos++ ); } // OK: I(this) ⪯ I(get) = ReadOnly
    public @Mutable void remove() { ArrayList.this.remove( this.currPos - 1 ); } // OK: I(this) ⪯ I(remove) = Mutable
  }
}
```

Figure 8. Declaration of the interface `Iterator` and the class `ArrayList` with an inner class `ArrItr`.

be illegal in IGJ. In the example above the immutability of the constructor must be readonly because `iterator()` is readonly.

2.7 Exceptions, immutable classes, reflection, and arrays

In IGJ's syntax, the immutability is an integral part of the type. In Javari [32] (see Sec. 3) it is syntactically possible but semantically illegal to write this code:

```
class Cell<X> { readonly X x; ...}
```

It is semantically illegal because the immutability of `X` is determined in the client code, e.g., `Cell<readonly Date>`. In comparison, IGJ's syntax does not even enable such a declaration: it is syntactically and semantically illegal.

Throwable Generics and immutability naturally combine in another aspect: their *usage limitations*. For example, it is forbidden to throw a readonly reference because the catcher can mutate that reference. Similarly, Java prohibits adding generic parameters to any subclass of `Throwable` because the compiler cannot statically connect the throwing and catching positions. IGJ naturally inherits this usage limitation from the underlying generics mechanism.

Manifest classes and immutable classes IGJ supports *manifest classes* [27], which are classes without an immutability parameter. Manifest classes can be used to express *class immutability*, e.g.,

```
class String extends Object<Immutable> ...
```

IGJ treats all methods of `String` as if they were annotated with `@Immutable`, and issues errors if mutable methods exist.

Reflection It is discouraged in IGJ to use reflection or to remove the immutability parameter by casting to a raw type. The IGJ compiler issues a *warning* in both cases because they can create holes in the type system. (IGJ does not consider these *errors* because it might be necessary to call legacy code.)

Arrays Java does not permit arrays to have generic parameters. IGJ supplies a wrapper class `Array<I extends ReadOnly,T>` that enables the creation of immutable arrays. IGJ treats an array type `T[]` as `Array<Mutable,T>`, i.e., arrays are mutable by default.

3. Previous Work

We are not aware of any work that proposed a static typing system for *object immutability* and not just reference immutability. Pechtchanski and Sarkar [25] describe various annotations for immutability assertions, such as `@immutableField`, `@immutableParam`, etc., and show that such assertions enable optimizations that can speed up some benchmarks by 5–10%. However they do not present any typing rules to enforce such assertions.

Java already includes various classes whose instances are immutable, and it supports a non-transitive form of immutability using `final`[3]. Functional languages such as ML default all fields to being immutable, with mutable fields being the exception. C++'s `const` mechanism has similar semantics to IGJ: a field can be declared as `const` (similar to readonly in IGJ), `mutable`, or by default as `this`-mutable. However, in contrast to IGJ, parameters to functions, return types, local variables, and generic parameters cannot be `this`-mutable. Other disadvantages are: (i) `const` can be cast away at any time, making it more a suggestion than a binding contract, (ii) `const` protects only the state of the enclosing object and not objects it points too, e.g., you cannot mutate an element inside a `const` node in a list, but the `next` node is mutable, and (iii) using `const` results in code duplication such as two versions of `operator[]` in every collection class in the STL.

Most of IGJ terminology was borrowed from Javari [32] such as assignable, readonly, mutable, `this`-mutable. However, this-mutable fields in Javari are mutable as lvalue and readonly as rvalue. Javari does not support object immutability, whereas its reference immutability is more limited than that of IGJ for two reasons: (i) parameters to methods, return types, and local variables, cannot be `this`-mutable, and (ii) IGJ can use generics to abstract over the immutability parameter. Javari's `romaybe` is in essence a template over immutability. For instance, if you wish to find an edge in a graph, then the algorithm will be the same regardless if the graph contains mutable or readonly edges, however the return type changes depending on the immutability of the edges. In IGJ, instead of inventing a new template-like mechanism such as `romaybe`, we use generics to achieve the same goal, as demonstrated by the static method `findEdge` on line 17 of Fig. 4. Javari also supports `this`-assignable fields, which pass the assignability (`final` or `assignable`) of `this` to a field. Finally, Javari uses `?readonly` which is similar to Java's wildcards. Consider, for instance, the class `Foo` written in Javari's syntax:

```
class Foo { public List<Object> list; }
```

Then in a `readonly Foo` the type of `list` is

```
readonly List<?readonly Object>
```

which is syntactic sugar for

```
readonly List<? extends readonly Object
                super mutable Object>
```

Thus, it is possible to insert only mutable elements to `list`, and retrieve only readonly elements.

[3] A final field cannot be assigned a new value after the constructor finishes. However, this restriction can be circumvented using reflection.

Skoglund and Wrigstad [30] propose a system with read and write references with similar semantics to C++'s `const`. They also introduce a `caseModeOf` construct which permits run-time checking of reference writeability.

Several papers proposed a mechanism of *access rights*. JAC [22] is a compile-time access right system with the following access right order: `readnothing < readimmutable < readonly < writeable`, where `readnothing` cannot access fields of `this` (only the identity for equality), and `readimmutable` can only access immutable state of `this`. JAC uses additional keywords (such as `nontransferable`) that address other concerns than immutability. Capabilities for sharing [5] are intended to generalize various other proposals for access rights, ownership and immutability, by giving 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.

Boyland [4]. concludes that readonly is useful but it does not address observational exposure, i.e., modifications on one side of an abstraction boundary that are observable on the other. However, IGJ's immutable objects can be shared safely because their state cannot change. Boyland's second criticism was that the **transitivity** principle (see Sec. 1), which is mandatory in all the designs above, should be selectively applied by the designer, because, "the elements in the container are not notionally part of the container" [4]. In IGJ, a programmer can solve this problem by using a different immutability for the container and its elements.

Non-null types [16] has a similar challenge that IGJ has in constructing immutable objects: a partially-initialized object may escape its constructor. IGJ uses `@AssignsFields` to mark a constructor of immutable objects, and a partially initialized object can escape only as `ReadOnly`. Non-null types uses a `Raw` annotation *on references* that might point to a partially-initialized object, and *on methods* to denote that the receiver can be `Raw`. A non-null field of a `Raw` object has different lvalue and rvalue: it is possible to assign only non-null values to such field, whereas reading from such field may return `null`. Similarly to IGJ, non-null types cannot handle cyclic data-structures, nor can it express the staged initialization paradigm in which the construction of an object continues after its constructor finishes.

3.1 Ownership types and readonly references

Ownership types [3, 8, 23] impose a structure on the references between objects in a program's memory. Ownership-enabled languages such as Ownership Generic Java [27] prevent aliasing to the internal state of an object. While preventing exposure of owned objects, ownership does not address exposing immutable parts of an object that cannot break encapsulation.

One possible application of ownership types is the ability to reason about read and write effects [9] which has complimentary goals to object immutability. Universes [13] is a Java language extension combining *ownership and reference immutability*. Most ownership systems enforce that all reference chains to an owned object pass through the owner. Universes relaxes this demand by enforcing this rule only for mutable references, i.e., readonly references can be shared without restriction.

3.2 Covariant subtyping

Covariant subtyping is allowing covariant changes in generic parameters in a type-safe manner. Variant parametric types [20] attach a variance annotation to a type argument, e.g., `Vector<+Number>` or `Vector<-Number>`. The subtype relation contains the following chain:

`Vector<Integer>` $\preceq$ `Vector<+Integer>` $\preceq$
$\preceq$ `Vector<+Number>` $\preceq$ `Vector<+Object>`

The type checker prohibits calling `someMethod(X)` when the receiver is of type `Foo<+X>`. For instance, suppose there is a method `isIn(X)` in class `Vector<X>`. Then, it is prohibited to call `isIn(Number)` on a reference of type `Vector<+Number>`.

Java's wildcards have a similar chain in the subtype relation:

`Vector<Integer>` $\preceq$ `Vector<? extends Integer>` $\preceq$
$\preceq$ `Vector<? extends Number>` $\preceq$ `Vector<? extends Object>`

Java's wildcards and variant parametric types are different in the legality of invoking `isIn(? extends Number)` on a reference of type `Vector<? extends Number>`. While such an invocation is prohibited in variant parametric types, Java permits such an invocation, however the only value of type `? extends Number` is `null`.

IGJ also contains a similar chain:

`Vector<Mutable,Integer>` $\preceq$ `Vector<ReadOnly,Integer>` $\preceq$
$\preceq$ `Vector<ReadOnly,Number>` $\preceq$ `Vector<ReadOnly,Object>`

However the restriction on method calls in IGJ is based on *semantics* (whether the method is readonly or not) rather than on *method signature* as in wildcards and variant parametric types. For example, IGJ allows calling `isIn(Number)` on a reference of type `Vector<ReadOnly, Number>` iff `isIn` is readonly.

3.3 Typestates for objects

In a typestates system, each object is in a certain state, and the set of applicable methods depends on the current state. Verifying typestates statically is challenging due to the existence of aliases, i.e., a state change in a particular object must affect all existing aliases to it. Typestates for objects [12] uses linear types to manage aliasing.

Object immutability can be partially expressed using typestates: by using two states (mutable and immutable) and declaring that mutating methods are applicable only in the mutable state. An additional method should mark the transition from a mutable state to an immutable state, and it should be called after the initialization of the object has finished. It remains to be seen if systems such as [12] can handle arbitrary aliases that occur in real programs, e.g., `this` references that escape in the constructor.

4. Experimentation

To illustrate the usability of IGJ we converted the programs of the jolden benchmark to IGJ, using an Eclipse plug-ins. We compiled those programs using the IGJ compiler, and we examined the code to find places that IGJ could provide stronger immutability guarantees.

Converting Java to IGJ We have created two Eclipse plug-ins for converting Java code into IGJ. The first plug-in converts full classes into IGJ by adding type parameters (`<I extends ReadOnly >`) to each class, and setting the immutability of fields, locals, and method to `Mutable`. The second plug-in generates IGJ skeletons of libraries' public signatures. This allows the developer to use or change the immutability interface of library classes, while avoiding the need to modify the library code to reflect those changes.

Due to Java restrictions, type parameters are not added in the following cases:

1. Accessing static fields or methods
2. In the context of reflection (`.class` or `instanceof`)
3. Array construction

```
 1: public static QuadTreeNode
 2:      createTree(QuadTreeNode parent,...)  {
 3:    QuadTreeNode node;
 4:    if(...)  { node = new BlackNode(...); }
 5:    else if(...)  { node = new WhiteNode(...); }
 6:    else {
 7:      node = new GreyNode(...);
 8:      sw = createTree(node, ...);
 9:      se=...; nw=...; ne=...;
10:      node.setChildren(sw,se,nw,ne);
11:    }
12:    return node;
13: }
```

Figure 9. The `QuadTreeNode.createTree` method. This method creates the `QuadTreeNode` corresponding to a given image. When the class `QuadTreeNode` is immutable, the last call to the `setChildren` method fails.

4. String due to string literals, and boxed classes (Integer, Boolean, ...) due to auto boxing.
5. Exceptions

IGJ compiler The IGJ compiler is an extension to Sun's javac. During the type checking phase of the Java language implementation, a visitor pattern is used to visit every element in the Abstract Syntax Tree (AST). IGJ uses two additional visitors for the AST: one to visit before the Java attribution phase, and one to visit after the Java attribution phase. The first visitor checks for appropriate use of the immutability parameter. The second visitor detects any violation of the typing rules. Finally, `isSubType` was modified according to Def. 2.1.

Example The perimeter program from jolden computes the perimeter of a region in a binary image represented by a quad-tree. This program has ten classes in three hierarchies. We annotated the `Quadrant` and `QuadTreeNode` hierarchies as being made up of immutable classes: we made those classes extend `Object<ReadOnly>`, we set the immutability for all methods and fields to `ReadOnly`, and we annotated the constructors (and a `setChildren` method called by one of them) as `AssignsFields`.

We had to perform one refactoring to solve a problem in the `createTree` static method (Fig. 9). This method creates the `QuadTreeNode` representing the image given as a parameter. The type of the created node is determined by the parameters. In the case of a `GreyNode` on Line 7, the method recursively constructs four new nodes representing the northwest, northeast, southwest, southeast quadrants of the image. After the four nodes are created, the method `setChildren` is called to set the four constructed nodes.

This code doesn't compile in IGJ. Because `QuadTreeNode` is an immutable class, it is illegal to call the mutating method `setChildren` (line 10). To solve this problem we refactored the code by moving the construction of the image quadrants into the constructor of `GreyNode`.

5. Proof of Type Soundness

Proving soundness is essential in face of complexities such as the new subtype definition (Def. 2.1) and mutable fields (Sec. 2.5). This section gives the type rules of a simplified version of IGJ and proves property (1) from Sec. 2.4. We are not aware of any previous work that proved a reference immutability theorem such as "readonly references cannot be converted to mutable". Property (1) implies such theorem, or else it would be possible to convert immutable to readonly, and than to mutable.

Our type system, called Featherweight IGJ (FIGJ), is based on Featherweight Generic Java (FGJ) [19]. FIGJ models the essence of IGJ: the fact that only mutable references can assign to fields, and the new subtype definition. Similar to the way FGJ removed many features from Java (such as null values, assignment, overloading, private, etc.), we removed from IGJ all method annotations. In other words, all methods are readonly, with the exception of a single constructor that assigns its arguments to the object's fields, thus making `AssignsFields` redundant. Assignment must be done from the "outside", i.e., instead of calling a setter method we must set the field from the outside (all fields are considered public in FGJ).

Finally, we restrict each class in FIGJ to have a *single* immutability parameter which extends `ReadOnly`, i.e., FIGJ cannot express manifest classes such as `String`.

Sec. 5.1 describes the syntax of FIGJ. The FIGJ subtype relation is presented in Sec. 5.2. Sec. 5.3 modifies the FGJ type checking rules to allow field assignment only by a mutable reference. Sec. 5.4 modifies FGJ operational semantics (reduction rules) to get stuck when assigning to a field of an immutable object, and Sec. 5.5 proves that in a well-typed program this never happens.

5.1 Featherweight IGJ Syntax

Because FIGJ models immutability, we had to add imperative extensions to FGJ such as assignment to fields, object locations and a store [26]. We also add three special types: `ReadOnly`, `Immutable`, and `Mutable`.

Fig. 10 presents the syntax of FIGJ. It defines types (`T`), non-variable types (`N`), immutability parameters (`I`), class declarations (`L`), method declarations (`M`), and expressions (`e`). Expressions in FIGJ include the five expressions in FGJ (method variable, field and method access, new instance creation, and cast), as well as our imperative extensions (field update and locations). Note that an immutability parameter is not a type in `N`, and the only places it appears in the syntax is as the first generic parameter of a type and the first generic variable of a class. The root of the class hierarchy is $\mathtt{Object}\texttt{<}\mathtt{X} \lhd \mathtt{ReadOnly}\texttt{>}$.

To support field assignment we define a store $\mathtt{S} = \{\mathtt{l} \mapsto \mathtt{N(l)}\}$ that maps locations to created objects. Note that we do not need a store typing [26] because the store already contains the type of each location. For a simpler notation we use a single symbol Δ to denote an environment that maps (i) variables to their types, (ii) type variables to their bounds (which are non-variable types): $\Delta = \{\mathtt{x} : \mathtt{T}\} \cup \{\mathtt{X} \preceq \mathtt{N}\}$. (The notation used in [26] is: typing context Γ, environment Δ.)

The field, method type, and method body lookup functions are based on their counterparts in FGJ and thus omitted from this paper. We define an additional auxiliary function that returns the immutability parameter $I(\mathtt{C}\texttt{<}\mathtt{X},\mathtt{X}\texttt{>}) = \mathtt{X}$.

Fig. 11 defines which type-variables are no-variant. We use two auxiliary functions: (i) $FV(\mathtt{T})$ is the set of free variables in `T`, (ii) $subterm(\mathtt{N})$ is the set of all subterms of types in $fields(\mathtt{N})$. (We do not need to consider the superclass as in NoVariant Rule because *fields* includes all fields of the superclass as well.)

After *NoVariant* reaches a fixed point on the class declarations, we can define $CoVariant(\mathtt{X}_i, \mathtt{C}\texttt{<}\mathtt{X}\texttt{>})$ to be the negation of *NoVariant*. In order for the class declarations to be wellformed, the immutability parameter must always be in *CoVariant*:

$$CoVariant(\mathtt{X}_1, \mathtt{C}\texttt{<}\mathtt{X}\texttt{>}) \text{ for any class } \mathtt{C}. \qquad (2)$$

We make the same assumptions as in FGJ about the correctness of the class declarations (e.g., that there are no circles in the subclass relation, that we have no method overloading, etc). We also use the same judgements as in FGJ, such as type, store, expressions,

$\mathtt{T} ::= \mathtt{X} \mid \mathtt{N}$	Type.
$\mathtt{N} ::= \mathtt{C<I, T>}$	Non-variable type.
$\mathtt{I} ::= \mathtt{ReadOnly} \mid \mathtt{Mutable} \mid \mathtt{Immutable} \mid \mathtt{X}$	Immutability parameter.
$\mathtt{L} ::= \mathtt{class}\ \mathtt{C<X} \lhd \mathtt{ReadOnly, X} \lhd \mathtt{N>} \lhd \mathtt{C'<X, T'>}\{\mathtt{T\ f;\ M}\}$	Class declaration.
$\mathtt{M} ::= \mathtt{<X} \lhd \mathtt{N>\ T\ m(T\ x)}\ \{\mathtt{return\ e;}\}$	Method declaration.
$\mathtt{e} ::= \mathtt{x} \mid \mathtt{e.f} \mid \mathtt{e.m<T>(e)} \mid \mathtt{new\ N(e)} \mid \mathtt{(N)\ e} \mid \mathtt{e.f} = \mathtt{e} \mid \mathtt{l}$	Expressions.

Figure 10. FIGJ Syntax.

$$\frac{\mathtt{C'<Mutable, T>} \in \mathit{subterm}(\mathtt{C<X>}) \qquad \mathtt{X}_i \in \mathit{FV}(\mathtt{T}_j)}{\mathit{NoVariant}(\mathtt{X}_i, \mathtt{C<X>})} \quad \text{(MC1)}$$

$$\frac{\mathtt{C'<T>} \in \mathit{subterm}(\mathtt{C<X>}) \qquad \mathit{NoVariant}(\mathtt{Y}_j, \mathtt{C'<Y>}) \qquad \mathtt{X}_i \in \mathit{FV}(\mathtt{T}_j)}{\mathit{NoVariant}(\mathtt{X}_i, \mathtt{C<X>})} \quad \text{(MC2)}$$

Figure 11. Definition of $\mathit{NoVariant}(\mathtt{X}_i, \mathtt{C<X>})$.

method and class wellformedness, with minor differences (e.g., instead of Object we use Object<I >).

FIGJ is more strict than FGJ regarding method overriding because FIGJ requires that the erased signatures of methods be identical (up to renaming of type variables):

Definition 5.1. *The* erased signature *of method* m(T x) *in class* $\mathtt{C<X} \lhd \mathtt{N>}$ *is*

$\lambda\mathtt{X}.[\mathtt{N}/\mathtt{X}]\mathtt{T}$

5.2 Subtyping

Fig. 12 shows FIGJ subtyping rules. The first four rules are the same as FGJ rules. Additionally, two special classes — Mutable and Immutable— are considered a separate class hierarchy extending ReadOnly. The rule S1 is a formalization of Def. 2.1.

Observe that the subtype relation is reflexive and transitive. Note that from rule S1 and (2) we have that

if $\mathtt{I} \preceq \mathtt{I}'$, then $\mathtt{C<I, T>} \preceq \mathtt{C<I', T>}$.

We write $\mathtt{T} \preceq \mathtt{T}'$ as a shorthand for $\Delta \vdash \mathtt{T} \preceq \mathtt{T}'$. Observe in rule S1 that the requirement $\mathtt{Immutable} \preceq \mathtt{T}'_1$ is equivalent to $\mathtt{T}'_1 \neq \mathtt{Mutable}$ and $\mathtt{T}'_1 \neq \mathtt{X}$ (where X is a variable), because an immutability parameter can have only four values according to the syntax rule for I in Fig. 10.

In Java, if $\mathtt{A} \preceq \mathtt{B}$ and f is a field of B, then the type of field f in A and B is exactly the same, i.e., $\mathtt{A.f} = \mathtt{B.f}$. In IGJ, we revise this property and demand that $\mathtt{A.f} \preceq \mathtt{B.f}$. For instance, consider the field edges on line 12 of Fig. 4, and the following two variables:

```
Graph<Mutable> mutG;
Graph<ReadOnly> roG;
// mutG.edges has the type List<Mutable,Edge<Mutable>>
// roG.edges has the type List<ReadOnly,Edge<ReadOnly>>
```

And indeed the first is a subtype of the latter.

Lemma 5.2. *Let* $\mathtt{T} \preceq \mathtt{T}'$*,* $\mathtt{F}'\ \mathtt{f} \in \mathit{fields}(\mathtt{T}')$*. Then* $\mathtt{F}\ \mathtt{f} \in \mathit{fields}(\mathtt{T})$ *and* $\mathtt{F} \preceq \mathtt{F}'$*.*

Proof. It is trivial to prove that field f exists in *fields*(T). We will prove that $\mathtt{F} \preceq \mathtt{F}'$ by induction on the derivation of $\mathtt{T} \preceq \mathtt{T}'$. Consider the last rule in the derivation sequence. The proof for the first six rules is immediate from the definition of *fields* and the fact that subtyping is transitive.

Now consider rule S1, where $\mathtt{T} = \mathtt{C<U>}$,and $\mathtt{T}' = \mathtt{C<U'>}$:

$\mathtt{U}_i = \mathtt{U}'_i$ or

$(\mathtt{Immutable} \preceq \mathtt{U}'_1$ and $\mathtt{U}_i \preceq \mathtt{U}'_i$ and $\mathit{CoVariant}(\mathtt{X}_i, \mathtt{C<X>}))$

Let V denote the type of field f in C<X>. Then we have that $\mathtt{F} = [\mathtt{U}/\mathtt{X}]\mathtt{V}$ and $\mathtt{F}' = [\mathtt{U}'/\mathtt{X}]\mathtt{V}$. On the one hand, if $\mathtt{U}_i = \mathtt{U}'_i$ for all i, then $\mathtt{T} = \mathtt{T}'$ and thus $\mathtt{F} = \mathtt{F}'$.

On the other hand, for some value i, we have that $\mathtt{U}_i \neq \mathtt{U}'_i$. Therefore we know from rule S1 that $\mathtt{Immutable} \preceq \mathtt{U}'_1$, $\mathtt{U}_i \preceq \mathtt{U}'_i$ and $\mathit{CoVariant}(\mathtt{X}_i, \mathtt{C<X>})$. Let i range over the integers in which $\mathtt{U}_i \neq \mathtt{U}'_i$. We will prove by induction that for every subterm A<S> in V, we have $[\mathtt{U}_i/\mathtt{X}_i]\mathtt{A<S>} \preceq [\mathtt{U}'_i/\mathtt{X}_i]\mathtt{A<S>}$. From rule S1 we need to prove that for all j: either $[\mathtt{U}_i/\mathtt{X}_i]\mathtt{S}_j = [\mathtt{U}'_i/\mathtt{X}_i]\mathtt{S}_j$ or,

$\mathtt{Immutable} \preceq [\mathtt{U}'_i/\mathtt{X}_i]\mathtt{S}_1$ and $[\mathtt{U}_i/\mathtt{X}_i]\mathtt{S}_j \preceq [\mathtt{U}'_i/\mathtt{X}_i]\mathtt{S}_j$ and $\mathit{CoVariant}(\mathtt{Y}_j, \mathtt{A<Y>})$

If $\mathtt{X}_i \notin \mathit{FV}(\mathtt{S}_j)$, then $[\mathtt{U}_i/\mathtt{X}_i]\mathtt{S}_j = [\mathtt{U}'_i/\mathtt{X}_i]\mathtt{S}_j$. Thus assume that $\mathtt{X}_i \in \mathit{FV}(\mathtt{S}_j)$.

If $\mathtt{S}_1 = \mathtt{X}_1$, then $[\mathtt{U}'_i/\mathtt{X}_i]\mathtt{S}_1 = \mathtt{U}'_1$ and we already showed that $\mathtt{Immutable} \preceq \mathtt{U}'_1$. Otherwise $[\mathtt{U}'_i/\mathtt{X}_i]\mathtt{S}_1 = \mathtt{S}_1$, and we will show that $\mathtt{Immutable} \preceq \mathtt{S}_1$ by showing that $\mathtt{S}_1 \neq \mathtt{Mutable}$ and $\mathtt{S}_1 \neq \mathtt{I}$. According to rule MC1 in Fig. 11, if $\mathtt{S}_1 = \mathtt{Mutable}$, then $\mathit{NoVariant}(\mathtt{X}_i, \mathtt{C<X>})$, which is a contradiction. We also have that $\mathtt{S}_1 \neq \mathtt{I}$ because from $\mathtt{Immutable} \preceq \mathtt{U}'_1$ we have that $\mathtt{U}'_1 \neq \mathtt{I}$ and thus V cannot contain I as a free variable. From the induction we have $[\mathtt{U}_i/\mathtt{X}_i]\mathtt{S}_j \preceq [\mathtt{U}'_i/\mathtt{X}_i]\mathtt{S}_j$. Finally we have that $\mathit{CoVariant}(\mathtt{Y}_j, \mathtt{A<Y>})$, because, if $\mathit{NoVariant}(\mathtt{Y}_j, \mathtt{A<Y>})$, according to rule MC2 in Fig. 11, we have the contradiction that $\mathit{NoVariant}(\mathtt{X}_i, \mathtt{C<X>})$. □

It is easy to add covariant fields to FGJ without breaking type-safety, because FGJ that does not include field assignment. Lem. 5.3 proves that whenever it is possible to assign to a field, it is never covariant. Intuitively, in a legal assignment $\mathtt{e}_1\mathtt{.f} = \mathtt{e}_2$, where $\mathtt{e}_1 : \mathtt{T}$, we have that $I(\mathtt{T}) = \mathtt{Mutable}$, and we will show that in all subtypes $\mathtt{T}' \preceq \mathtt{T}$ the field f have the same type.

Lemma 5.3. *Let* $\Delta \vdash \mathtt{T} \preceq \mathtt{T}'$*,* $\mathtt{F}'\ \mathtt{f} \in \mathit{fields}(\mathit{bound}_\Delta(\mathtt{T}'))$*,* $\mathtt{F}\ \mathtt{f} \in \mathit{fields}(\mathit{bound}_\Delta(\mathtt{T}))$*, and* $I(\mathtt{T}') = \mathtt{Mutable}$*. Then* $\mathtt{F} = \mathtt{F}'$*.*

Proof. By induction on the derivation of $\Delta \vdash \mathtt{T} \preceq \mathtt{T}'$, similarly to Lemma A.2.8 in [19]. Because $I(\mathtt{T}') = \mathtt{Mutable}$ we also have that $I(\mathtt{T}) = \mathtt{Mutable}$, and whenever rule S1 is applied, it can never be that $\Delta \vdash \mathtt{Immutable} \preceq \mathtt{T}'_1$, thus we need to consider the same set of subtyping rules as in FGJ. □

5.3 FIGJ Expressions

Fig. 13 presents the type checking rules in IGJ. Rule T-FIELD-SET ensures that only a mutable reference can assign to a field.

$$\frac{\Delta \vdash \mathtt{T}_1 \preceq \mathtt{T}_2 \quad \Delta \vdash \mathtt{T}_2 \preceq \mathtt{T}_3}{\Delta \vdash \mathtt{T}_1 \preceq \mathtt{T}_3} \qquad \frac{}{\Delta \vdash \mathtt{X} \preceq \Delta(\mathtt{X})} \qquad \frac{\mathtt{class\ C{<}X} \lhd \mathtt{N{>}} \lhd \mathtt{N}\ \{\ldots\}}{\Delta \vdash \mathtt{C{<}T{>}} \preceq [\mathtt{T}/\mathtt{X}]\mathtt{N}}$$

$$\frac{}{\Delta \vdash \mathtt{T} \preceq \mathtt{T}} \qquad \frac{}{\Delta \vdash \mathtt{Mutable} \preceq \mathtt{ReadOnly}} \qquad \frac{}{\Delta \vdash \mathtt{Immutable} \preceq \mathtt{ReadOnly}}$$

$$\frac{\mathtt{T}_i = \mathtt{T}'_i \text{ or } (\Delta \vdash \mathtt{Immutable} \preceq \mathtt{T}'_1 \text{ and } \Delta \vdash \mathtt{T}_i \preceq \mathtt{T}'_i \text{ and } \mathit{CoVariant}(\mathtt{X}_i, \mathtt{C{<}X{>}}))}{\Delta \vdash \mathtt{C{<}T{>}} \preceq \mathtt{C{<}T'{>}}} \quad \text{(S1)}$$

Figure 12. FIGJ Subtyping Rules.

$$\frac{\ldots \quad \Delta, \mathtt{S} \vdash \mathtt{e} : \mathtt{T} \quad I(\mathtt{T}) = \mathtt{Mutable}}{\Delta, \mathtt{S} \vdash \mathtt{e.f}_i = \mathtt{e}' : \mathtt{T}'} \quad \text{(T-FIELD-SET)}$$

Figure 13. FIGJ Expression Typing: we show only the modifications to standard typing rules

5.4 Operational Semantics

Fig. 14 shows FIGJ reduction rules. FIGJ only modifies R-FIELD-SET to make sure that only a mutable location can write to its fields. The FIGJ context reduction rules (or congruence rules) are standard and are omitted from this paper.

Consider the rules T-FIELD-SET in Fig. 13 and R-FIELD-SET in Fig. 14. The first rule checks at "compile-time" that only a mutable expression (or *reference* in Java's terminology) can assign to a field, whereas the second checks at "run-time" that only a mutable location (or *object* in Java's terminology) can be mutated by assigning to its fields. Note that only objects have an immutability at run-time, not references.

5.5 Type Soundness

The type preservation theorem proves that if any expression reduces to another expression, then the latter is always a subtype of the former.

Theorem 5.4. ***(Type Preservation)*** *If* $\Delta, \mathtt{S} \vdash \mathtt{e} : \mathtt{T}$ *and* $\mathtt{e}, \mathtt{S} \rightarrow \mathtt{e}', \mathtt{S}'$*, then* $\exists \mathtt{T}'$ *such that* $\Delta \vdash \mathtt{T}' \preceq \mathtt{T}$ *and* $\Delta, \mathtt{S}' \vdash \mathtt{e}' : \mathtt{T}'$.

Proof. Using structural induction on the reduction rules in Fig. 14, while maintaining the invariant on the store that if $\mathtt{S}[\mathtt{l}] = \mathtt{N(l)}$, $\mathtt{S}[\mathtt{l}_i] = \mathtt{N}_i(\ldots)$, and $\mathit{fields}(\mathtt{N}) = \mathtt{T\ f}$, then $\mathtt{N}_i \preceq \mathtt{T}_i$. The only subtlety is in R-FIELD-SET where we use Lem. 5.3 to prove that if the field was assigned, then it is no-variant. □

The progress theorem shows that FIGJ programs don't get "stuck" and any well typed FIGJ expression that does not contain free variables (closed) can be reduced to some location or contains a failed downcast.

Theorem 5.5. ***(Progress)*** *Suppose* `e` *is a closed well-typed expression. Then either* `e` *is a location, or it contains a failed downcast, or there is an applicable reduction rule that contains* `e` *on the left hand side.*

Proof. Using a case by case analysis of all possible expression types in Fig. 13. The only change from the proof in FGJ is the use of T-FIELD-SET to prove that $I(\mathtt{l}) = \mathtt{Mutable}$ in R-FIELD-SET, and therefore we never get stuck due to that rule. □

Thm. 5.6 is a formalization of property (1).

Theorem 5.6. *Let* $\Delta, \mathtt{S} \vdash \mathtt{e} : \mathtt{T}$*, and* $\mathtt{e}, \mathtt{S} \rightarrow^* \mathtt{l}, \mathtt{S}'$*, where* $\mathtt{S}'[\mathtt{l}] = \mathtt{N(l)}$*. Then* $\Delta \vdash I(\mathtt{N}) \preceq I(\mathtt{T})$.

Proof. From Thm. 5.4 we have $\Delta \vdash \mathtt{N} \preceq \mathtt{T}$, and thus $\Delta \vdash I(\mathtt{N}) \preceq I(\mathtt{T})$. □

6. Future Work

We plan to add several features to the IGJ compiler, such as: (i) a warning if `this` escapes (see Remark 2.2), (ii) field initializers in which `this` is mutable if all constructors are mutable, and (iii) a mechanism to suppress specific warnings such as `@SuppressWarnings("object-cast")`. Similarly to tools for inferring generic arguments, immutability annotations can also be produced automatically. A plugin for an IDE such as Eclipse may also increase adoption of IGJ. We mention four other areas for future work: a default for type-variables, an alternative syntax, runtime support, and a `WriteOnly` immutability parameter.

IGJ's syntax does not permit defining a default value for the immutability parameter. For example, in Javari fields are `this`-mutable by default, and everything else is mutable by default. We propose to extend Java syntax to allow declarations such as

```
class Graph<I extends ReadOnly default Mutable>
```

One drawback is that if a different default is chosen in a subclass, we can have non-intuitive errors. For instance, consider `Graph3D` which is by default immutable:

```
class Graph3D<I extends ReadOnly default Immutable>
    extends Graph<I > {...}
Graph g = ...;
if (g instanceof Graph3D) {
   Graph3D g3d = (Graph3D) g; } // error!
```

The reason for the error is that `g` is mutable whereas `g3d` is immutable.

A different alternative syntax can come from Java's annotations, however, currently, they cannot appear on any use of a type. JSR 308 [15] proposed for Java 7 allows annotations to be used practically anywhere, e.g., `@immutable Document[@readonly]` and `new @mutable ArrayList<@immutable Edge>(...)`. A type can have four types of annotations: `@Immutable`, `@Mutable`, `@ReadOnly`, and `@ThisMutable`. Instead of using a generic method to abstract over the immutability parameter (like in line 18 of Fig. 4), we can use `romaybe`.

Currently no representation of immutability exists at runtime. Java does not support checked casts involving generics, (e.g., casting a vector of objects to a vector of integers always succeeds), so safe down-casting is not feasible.

In Javari [32] the following mechanism is suggested to safely cast a readonly reference to mutable, while still preventing side effects through the mutable reference. we set a bit in that reference which is checked before every mutating operation. That bit is propagated during reference assignment and when copying a reference to a field.

A different runtime mechanism is using a *readonly proxy*, i.e., adding to `Object` the method:

```
@ReadOnly Object<Mutable> Object.createReadOnlyProxy()
```

That is similar in spirit to Java's methods

$$\frac{\texttt{l} \notin dom(\texttt{S}) \qquad \texttt{S}' = \texttt{S}[\texttt{l} \mapsto \texttt{N(l)}]}{\texttt{new}\ \texttt{N(l)}, \texttt{S} \longrightarrow \texttt{l}, \texttt{S}'} \quad \text{(R-NEW)}$$

$$\frac{\texttt{S}[\texttt{l}] = \texttt{N(l)} \qquad \mathit{fields}(\texttt{N}) = \overline{\texttt{T}}\ \overline{\texttt{f}}}{\texttt{l.f}_i, \texttt{S} \longrightarrow \texttt{l}_i, \texttt{S}} \quad \text{(R-FIELD)}$$

$$\frac{\texttt{S}[\texttt{l}] = \texttt{N(l)} \quad I(\texttt{N}) = \texttt{Mutable} \quad \mathit{fields}(\texttt{N}) = \overline{\texttt{T}}\ \overline{\texttt{f}} \quad \texttt{S}' = \texttt{S}[\texttt{l} \mapsto [\texttt{l}'/\texttt{l}_i]\texttt{N(l)}]}{\texttt{l.f}_i = \texttt{l}', \texttt{S} \longrightarrow \texttt{l}', \texttt{S}'} \quad \underline{\text{(R-FIELD-SET)}}$$

$$\frac{\texttt{S}[\texttt{l}] = \texttt{N}(\overline{\texttt{l}'}) \qquad \mathit{mbody}(\texttt{m<}\overline{\texttt{V}}\texttt{>}, \texttt{N}) = \overline{\texttt{x}}.\texttt{e}}{\texttt{l.m<}\overline{\texttt{V}}\texttt{>}(\overline{\texttt{l}}), \texttt{S} \longrightarrow [\overline{\texttt{l}}/\overline{\texttt{x}}, \texttt{l/this}]\texttt{e}, \texttt{S}} \quad \text{(R-METHOD)}$$

$$\frac{\texttt{S}[\texttt{l}] = \texttt{N(l)} \qquad \texttt{N} \preceq \texttt{P}}{\texttt{(P)l}, \texttt{S} \longrightarrow \texttt{l}, \texttt{S}} \quad \text{(R-CAST)}$$

Figure 14. FIGJ Reduction Rules (only R-FIELD-SET was modified)

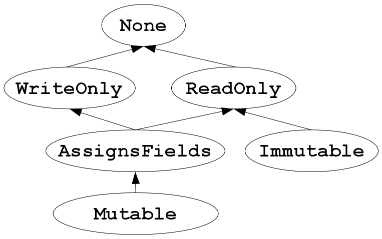

Figure 15. The type hierarchy for immutability parameters of Fig. 6 extended with `WriteOnly` and `None`.

`Collections.unmodifiable{Set,List}`, etc.

The IGJ compiler will automatically override this method in subclasses, covariantly changing the return type.The implementation will return a new mutable object, but all mutating methods are overridden to raise an exception. The compiler should issue a warning if any fields are public, because they can be mutated directly, bypassing the proxy protection.

Another challenge remaining is building an immutable cyclic data-structure, because that requires setting at least one reference after an immutable object has been created. A similar challenge exists in the staged initialization paradigm in which the object's construction continues after the constructor finished. Such challenges can be solved using a different runtime operation that converts a mutable object into an immutable one, i.e., not simply returning an immutable shallow proxy but modifying the object and all its fields which are `this`-mutable. Because it is impossible to invalidate all existing mutable references, the runtime system needs to modify the virtual table and replace mutating methods by methods that raise an exception (similarly to the proxy technique). In order to make this technique thread-safe, this operation will require similar synchronization as done at the end of a constructor (see Remark 2.2).

IGJ can be extended with a `WriteOnly` notation as well, as can be seen in Fig. 15. The parent of both `WriteOnly` and `ReadOnly` is `None` which denotes a type that have neither read nor write privileges. In a similar way that a readonly type can be changed co-variantly, a writeonly type can be changed contra-variantly.

Similar to the `@assignable` annotation for fields, we can add a `@readable` annotation to denote that a field can always be read, even in a writeonly type. The following example of a `Vector` class shows where `WriteOnly` can be used:

```
class Vector<I extends None, T> {
  @readable int size; ...
  @None int size() { return size; }
  @WriteOnly void add(T t) { ...}
  @WriteOnly void removeLast() { ...}
  @Mutable void remove(T t) { ...}
  @ReadOnly void get(int index) { ...}
}
```

Further research is needed to determine if the benefits outweigh the increased complexity.

7. Conclusions

This paper presented *Immutability Generic Java* (IGJ), a design for adding reference and object immutability on top of the existing generic mechanism in Java. IGJ satisfies the design principles in Sec. 1: transitivity, static checking, polymorphism, and simplicity. IGJ provides transitive immutability to protect the entire abstract state, and one can control this transitivity by using multiple immutability parameters, e.g., using different immutability for the container and its elements. IGJ is purely static, incurring no runtime penalties. IGJ does not lead to code duplication because it supports a high degree of polymorphism using covariant subtyping (Def. 2.1), and it is also possible to use generic methods to abstract over the immutability parameter. Finally, IGJ is simple, does not require changing Java's syntax, and has a small number of type-checking rules.

References

[1] S. Artzi, M. D. Ernst, A. Kieżun, C. Pacheco, and J. H. Perkins. Finding the needles in the haystack: Generating legal test inputs for object-oriented programs. In *M-TOOS*, Oct. 2006.

[2] A. Birka and M. D. Ernst. A practical type system and language for reference immutability. In *OOPSLA*, pages 35–49, Oct. 2004.

[3] C. Boyapati. *SafeJava: A Unified Type System for Safe Programming*. PhD thesis, EECS, MIT, February 2004.

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

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

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

[7] L. Burdy, Y. Cheon, D. Cok, M. D. Ernst, J. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of JML tools and applications. *STTT*, 7(3):212–232, June 2005.

[8] D. Clarke. *Object Ownership and Containment*. PhD thesis, School of CSE, UNSW, Australia, 2002.

[9] D. Clarke and S. Drossopoulou. Ownership, Encapsulation, and the Disjointness of Type and Effect. In *Proceedings of ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 292–310, Seattle, WA, USA, Nov. 2002. ACM Press, New York, NY, USA.

[10] L. R. Clausen. A Java bytecode optimizer using side-effect analysis. *Concurrency: Practice and Experience*, 9(11):1031–1045, 1997.

[11] V. Dallmeier, C. Lindig, A. Wasylkowski, and A. Zeller. Mining object behavior with ADABU. In *WODA*, pages 17–24, May 2006.

[12] R. DeLine and M. Fähndrich. Typestates for objects, June 2004.

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

[14] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE TSE*, 27(2):99–123, Feb. 2001.

[15] M. D. Ernst and D. Coward. JSR 308: Annotations on Java types. `http://jcp.org/en/jsr/detail?id=308`, Oct. 17, 2006.

[16] M. Fähndrich and K. R. M. Leino. Declaring and checking non-null types in an object-oriented language. In *OOPSLA*, pages 302–312, Nov. 2003.

[17] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 2000.

[18] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification*. Addison Wesley, Boston, MA, third edition, 2005.

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

[20] A. Igarashi and M. Viroli. Variant parametric types: A flexible subtyping scheme for generics. *ACM Trans. Program. Lang. Syst.*, 28(5):795–847, 2006.

[21] José Javier Dolado and Mark Harman and Mari Carmen Otero and Lin Hu. An empirical investigation of the influence of a type of side effects on program comprehension. *IEEE TSE*, 29(7):665–670, July 2003.

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

[23] Y. Lu and J. Potter. Ownership and accessibility. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP)*. Springer-Verlag, Berlin, Heidelberg, Germany, 2006.

[24] L. Mariani and M. Pezzè. Behavior capture and test: Automated analysis of component integration. In *ICECCS*, pages 292–301, June 2005.

[25] I. Pechtchanski and V. Sarkar. Immutability specification and its applications. In *Java Grande*, pages 202–211, Nov. 2002.

[26] B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[27] A. Potanin, J. Noble, D. Clarke, and R. Biddle. Generic ownership for generic Java. In *OOPSLA*, pages 311–324, Oct. 2006.

[28] W. Pugh. JSR 133: JAVA memory model and thread specification revision. `http://jcp.org/en/jsr/detail?id=133`, Sept. 30, 2004.

[29] A. Sălcianu. *Pointer analysis for Java programs: Novel techniques and applications*. PhD thesis, MIT Dept. of EECS, Sept. 2006.

[30] M. Skoglund and T. Wrigstad. A mode system for read-only references in Java. In *3rd Workshop on Formal Techniques for Java Programs*, June 18, 2001. Revised.

[31] O. Tkachuk and M. B. Dwyer. Adapting side effects analysis for modular program model checking. In *ESEC/FSE*, pages 188–197, Sept. 2003.

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

[33] T. Xie. Augmenting automatically generated unit-test suites with regression oracle checking. In *ECOOP*, pages 380–403, July 2006.