Immutability

Alex Potanin[1], Johan Östlund[2], Yoav Zibin[3], and Michael D. Ernst[4]

[1] School of Engineering and Computer Science, VUW, Wellington, New Zealand
[2] Department of Information Technology, Uppsala University, Uppsala, Sweden
[3] Google New York, NY, USA
[4] Computer Science and Engineering, University of Washington, WA, USA

Abstract. One of the main reasons aliasing has to be controlled, as highlighted in another chapter [1] of this book [2], is the possibility that a variable can unexpectedly change its value without the referrer's knowledge. This book will not be complete without a discussion of the impact of *immutability* on reference-abundant imperative object-oriented languages. In this chapter we briefly survey possible definitions of immutability and present recent work by the authors on adding immutability to object-oriented languages and how it impacts aliasing.

1 Introduction

Traditional imperative object-oriented (OO) programs consist of objects that have *state* and *behaviour*[1]. The behaviour is modelled by the methods. The state is represented by the values of an object's fields — which can either be primitives or references to other objects. *Immutable objects* are those whose state does not change after they are initialised [3–12].

Immutability information is useful in many software engineering tasks, including modeling [13], verification [14], compile- and run-time optimizations [15, 5, 16], program transformations such as refactoring [17], test input generation [18], regression oracle creation [19, 20], invariant detection [21], specification mining [22], and program comprehension [23]. The importance of immutability is highlighted by the documentation of the `Map` interface in Java that states: "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."

The notion of immutability is not as straightforward as it might seem, and many different definitions of immutability exist. Immutability may be deep (transitive) or shallow. In deep immutability, every object referred to by an immutable object must itself be immutable; in shallow immutability, the object's fields cannot be reassigned, but their referents may be mutated. Immutability guarantees may hold for every field of an object, or may exclude certain fields such as those used for caching. Immutability may be abstract or concrete. Concrete immutability forbids any change to an object's in-memory representation; abstract immutability permits benevolent side effects that do

[1] For example: `http://docs.oracle.com/javase/tutorial/java/concepts/object.html`.

not affect the abstraction, such as caching values, rebalancing trees, etc. Immutability guarantees may hold immediately or may be delayed. An immediate guarantee holds as soon as the constructor completes; a delayed guarantee permits initialization of a data structure to continue, effectively modifying an immutable object, which is necessary to allow circular initialisation for immutable data structures. Immutability guarantees can be about objects or references. A read-only reference cannot be used for mutation, but the underlying object can be mutated by an aliasing reference; an immutable object is never changed by any reference to it. This paper discusses these and other issues, and presents language designs that address them.

Outline. Section 2 introduces the rich concept of immutability and its variations. Section 3 outlines the current state of immutability support in object-oriented programming languages. Section 4 provides a number of motivations and describes advantages of proposed ways to support immutability mostly on top of a Java-like language. Section 5 quickly tours through the major recent proposals for adding immutability while extracting common themes and highlighting the differences. Section 6 discusses the immutability concept in more general setting, and Section 7 concludes.

2 What Is Immutability?

An immutable program component remains the same over time. Equivalently, changes to an immutable program component are forbidden.

An immutable object (Section 2.1) never changes. By contrast, a read-only reference (Section 2.2) cannot be used for mutation, but the referred-to object might change via other references. Assignment of fields/variables (Section 2.3) is not a mutation of the referred-to object but is sometimes confused with it.

Currently, there is no support in Java (or any mainstream OO language) to express and check object immutability or read-only references. Rather, programmers must use external tools that add these capabilities, or else resort to manual inspection.

2.1 Immutable Objects

An *immutable object* [11, 12] cannot be modified.

When every object of a given class is immutable, then we say the class is immutable. Examples of immutable classes in Java [24] include `String` and most subclasses of `Number`[2] such as `Integer` and `BigDecimal`. An immutable class contains no mutating methods that update/modify the receiver; rather, if a different value is required, a client calls a constructor or producer method that returns a new object. In addition to not providing mutating methods, all fields must be hidden from clients (e.g. made private).

Even if a class is not immutable, specific objects of that class may be immutable [11, 12]. For example, some instances of `List` in a given program may be immutable, whereas others can be modified. Here is example Immutability Generic Java (IGJ) [10] code that instantiates the same class `LinkedList` as both *mutable* and *immutable* object:

[2] `java.util.concurrent.AtomicInteger` is mutable, though.

```
LinkedList<Mutable> lm = new LinkedList<Mutable>();
LinkedList<Immutable> li = new LinkedList<Immutable>();
```

Object `lm` can be changed or mutated, for example by adding or removing elements. By contrast, `li` cannot be changed or mutated, even though it is implemented by the same `LinkedList` code.

2.2 Read-Only References

A *read-only* reference[3] [3–9] cannot be used to modify its referent. However, there may exist mutable aliases to the object elsewhere in the system. In other words, normal references carry the privilege to mutate the referent, and read-only references do not.

Usually a read-only reference's type is a supertype of a mutable reference's type, so a mutable reference can be used in any context in which a read-only one is legal. For example, continuing the previous list example, this is legal:

```
LinkedList<ReadOnly> lro = lm;
```

Note that `lm` is a mutable alias that can be used to mutate the list.

Since read-only references do not preclude the existence of mutable aliases, read-only references do not guarantee object immutability, unless read-only references are combined with an alias/escape analysis to guarantee that no mutable aliases to an object exist [25, 3].

A *const* pointer in `C++` is a shallow read-only reference.

Read-Only Method Parameters. If a method does not mutate one of its formal parameters, then that formal parameter can be annotated as read-only. Then, it is legal to call the method using a mutable, immutable, or read-only reference as the actual argument. If a method does mutate a formal parameter, then the method can only be called by passing in a mutable object as an argument. The receiver is treated the same as the other formal parameters.

Pure Methods. A *pure* method [26–29] has no externally-visible *side effects*. In other words, calling a pure method is guaranteed to leave every existing object in an unchanged state.[4] This is a stronger guarantee than asserting that every method parameter is (deeply) read-only, since it applies to static variables as well. For example:

```
@Pure
boolean has(String x) {
  for (String i : items) {
```

[3] "Reference immutability" is another standard term, but we use "read-only reference" to avoid potential confusion if a reader mis-interprets "reference immutability" as stating that the reference itself is immutable. Also see Section 2.3.

[4] Just as there are multiple varieties of immutability, there are multiple varieties of purity. Different definitions forbid all mutations, or permit only mutations of object allocated after the method is entered, or permit benevolent side effects on previously-existing objects.

```
    if (x == i) { return true; }
  }
  return false;
}
```

2.3 Non-assignability

Assignment is a property of a variable: it indicates whether the variable is permitted to be reassigned. Assignment of a variable is unrelated to mutation. In particular, no object is mutated in this code:

```
Date myVar = ...; // local variable
...
myVar = anotherDate;
```

Assignment of a field is a mutation of the object that contains the field, but is *not* a mutation of either the object that was previously in the field or of the object that is subsequently in the field. For example, in

```
myClass.itsDate = anotherDate;
```

no `Date` object has been mutated.

The `final` keyword in Java [24] prohibits assignment but not mutation. In the following example, the variable `v` is declared `final` and thus it cannot be reassigned after the declaration, though its value can be mutated:

```
final Foo v = new Foo(); // local variable
...
v = new Foo(); // compile−time error: assignment is forbidden
v.mutate(); // OK: mutation is permitted
```

2.4 Deep vs. Shallow Immutability; Abstract vs. Concrete Immutability

When specifying an immutability property, it is necessary to state whether the property is deep or shallow, and which fields of the object's representation are relevant.

Immutability and read-only references may be deep or shallow, depending on whether transitively-referred to objects are also required to be immutable. In deep immutability, it is forbidden not only to re-assign an object's fields, but also to mutate them. In shallow immutability, it is forbidden to re-assign an object's fields, but permitted to mutate them. Consider the following example:

```
class C {
  D f;
}
class D {
  int x;
}

```

```
C<Immutable> myC = ...;
...
myC.f = otherD; // illegal under both deep and shallow immutability
myC.f.x++; // legal under shallow immutability, illegal under deep
```

Most often, a client desires a deep rather than a shallow immutability guarantee. An OO program's representation of some object or concept in the real world often spans multiple objects in the program. As a simple example, a list may be represented by many `Link` objects connected in a linked list. A client does not know or care about the specific data representation, but wants a guarantee of immutability of the abstract value that the concrete data represents.

An orthogonal axis of immutability is which fields should be considered as protected by the immutability guarantee. A *benevolent side effect* is one that changes an object's representation, but does not change the object's abstract value. A common example is filling in a field that caches a value. Another example is the move-to-front optimization that speeds up looking up elements in a set that is represented as a list. Thus, it is possible that a change to an object's representation is *not* a change to the object's abstract value.

Similarly, just like with ownership-like schemes [30], it might make sense to make only part of an object deeply immutable (e.g. the fields specific to its *representation*) while keeping the other fields mutable. For example, an immutable list might contain mutable elements.

Most often, a client is concerned with the abstract value rather than details of the object's representation such as cached values or the order of objects in a set. However, reasoning about low-level properties such as interactions with the memory system may require a guarantee of *representation immutability* rather than *abstract immutability*.

3 Immutability in the Mainstream Programming Languages

In non-functional object-oriented languages immutability support is extremely limited. The only clear examples are the use of `const` in C++ and the use of the `final` keyword in Java, which we shall see are not enough to guarantee immutability of objects. We discuss existing support in popular languages and in the following Section we look into how recent proposals improve the state of affairs.

3.1 C++ `const`

C [31] and C++ [32] provide a `const` keyword for specifying immutability. C++'s `const` keyword is more commonly used as an aid when declaring interfaces, rather than as a way of declaring symbolic constants [32]. Furthermore, there are a number of pitfalls that led Java's designers to omit `const`.

Because of numerous loopholes, the `const` notation in C++ does not provide a guarantee of immutability even for accesses through the `const` reference. First, an unchecked cast can remove `const` from a variable. Second, C++'s `const_cast` may also be applied arbitrarily and is not dynamically checked. The `const_cast` operator was added to C++ to discourage, but not prohibit, use of C-style casts, accidental use of which may convert a read-only pointer or reference to a mutable one. Third, because C++ is not a type safe language, one can (mis)use type system weaknesses such as unions and varargs (unchecked variable-length procedure arguments) to bypass the restrictions on mutability prescribed by `const`.

Another criticism of C++'s `const` is that C++ does not permit parameterization of code based on the immutability of a variable. Use of `const` may lead to code duplication, where several versions of a function are needed depending on `const`-ness. An example is the two versions of `strchr` in the C++ standard library.

Finally, declaring a method as `const` (or read-only) only stops it from modifying the receiver and does not prevent it from modifying any other objects. Thus, a `const` (read-only) method in C++ is not a *pure* method.

C++'s `const` is shallow with respect to pointers but deep with respect to fields. C++ permits the contents of a read-only pointer to be modified, and read-only methods protect only the local state of the enclosing object. To guarantee transitive non-mutability, an object's state must be (transitively) held directly in variables/fields rather than accessed by a pointer. However, this precludes sharing, which is a serious disadvantage. Additionally, whereas C++ permits specification of `const` at each level of pointer dereference, it does not permit doing so at each level of a multi-dimensional array.

Most C++ experts advocate the use of `const` (for example, Meyers advises using `const` wherever possible [33]). However, as with many other type systems (including those of C++ and Java), some programmers feel that the need to specify types outweighs the benefits of type checking. At least three studies have found that static type checking reduces development time or errors [34–36]. We are not aware of any empirical (or other) evaluations regarding the costs and benefits of immutability annotations.

A common criticism of `const` is that transforming a large existing codebase to achieve `const` correctness is difficult, because `const` pervades the code: typically, all (or none) of a codebase must be annotated. This propagation effect is unavoidable when types or externally visible representations are changed. Inference of `const` annotations (such as that implemented by Foster et al. [37]) eliminates such manual effort. Even without a type inference, some [6] found the work of annotation to be greatly eased by fully annotating each part of the code in turn while thinking about its contract or specification, rather than inserting partial annotations and attempting to address type checker errors one at a time. The proper solution, of course, is to write `const` annotations in the code from the beginning, which takes little or no extra work, as the designer should have already made decisions about mutability.

3.2 Java **`final`**

Java [24] does not support `const`. Instead a `final` keyword was introduced. Java does not have a concept of "`const` references", and so there is no support for final methods, in the sense that only such methods would be invokable on final receivers. (In Java, `final` applied to a method means something entirely different: the method cannot be overridden in a subclass.) Similar to C++, marking a member as `final` only protects the variable, not the object the variable refers to, from being mutated. Thus, immutability in C++ and Java is not *transitive*.

3.3 Immutability in Non-Object-Oriented Languages

Functional languages, such as ML [38], default all variables to being immutable. OCaml [39] combines object-orientation with a `mutable` annotation on fields (for example, references are implemented as a one-compartment mutable record). However, there is little support from type systems to distinguish *mutable* operations from *read-only* operations.

4 How Can We Improve on the State of the Art?

The two major improvements achieved by the immutability proposals of the last decade are: (1) support for *transitive* read-only by supporting the enforcement of not just the reference to an object but also any further references from such object being read-only and (2) support for *object immutability* rather than just read-only references thus guaranteeing that no unexpected *alias* to an immutable object can change its state. Javari [6] supports the former, while Joe$_3$ [12] and OIGJ [11] support both.

To achieve read-only references, Javari utilizes additional annotations on any variable declaration in Java (e.g. `readonly`) that is then checked by the type system to guarantee that no read-only reference is assigned to or modified. Javari's implementation in Java is available for download[5].

Joe$_3$ utilizes an ownership-based type system and a very simple effect system which keeps track of which object is modified and prevents any modifications to objects which are *immutable* or *read-only*. A Polyglot-based[6] prototype implementation is available.

OIGJ makes use of Java's generic types to allow the types to state whether the object is *mutable* or *immutable* when creating the object. OIGJ also supports *read-only* types. A Checker-Framework-based[7] prototype implementation is available.

This section presents examples taken from three recent immutability proposals by the authors. Subsection 4.1 presents enforcement of contracts and read-only access to internal data in the Javari system [6] which supports read-only references. Subsection 4.2 presents a larger example similar to `LinkedList` from the Java collections written in the OIGJ [11] system supporting both read-only references and object immutability, among other features. Subsection 4.3 shows how to support flexible lists and context-based read-only in Joe$_3$ [12], a system supporting object immutability and more. Both OIGJ and Joe$_3$ make use of *ownership* [40] to be able to properly support deep immutability as discussed later in this chapter. Please refer to Figure 7 for a summary of supported features in the abovementioned systems and several others.

4.1 Cases from Javari

This subsection shows examples of read-only as found in the Javari system.

[5] `http://types.cs.washington.edu/javari/`
[6] `http://www.cs.cornell.edu/projects/polyglot/`
[7] `http://types.cs.washington.edu/checker-framework/`

Enforcement of contracts. Consider a voting system containing the following routine:

```
ElectionResults tabulate(Ballots votes) { ... }
```

In order to permit verification (e.g. double checking) of the results, it is necessary to safeguard the integrity of the ballots. This requires a machine-checked guarantee that the routine does not modify its input `votes`. Using Javari, the specification of `tabulate` could declare that `votes` is read-only:

```
ElectionResults tabulate(readonly Ballots votes) {
  ··· //cannot tamper with the votes
}
```

and the compiler ensures that implementers of `tabulate` do not violate the contract.

Read-only access to internal data. Accessor methods often return data that already exists as part of the representation of the module.

For example, consider the `Class.getSigners` method, which returns the entities that have digitally signed a particular implementation. In JDK 1.1.1, its implementation is simple and efficient:

```
class Class {
  private Object[] signers;
  Object[] getSigners() {
    return signers;
  }
}
```

This is a security hole, because a malicious client can call `getSigners` and then add elements to the `signers` array.

Javari permits the following solution:

```
class Class {
  private Object[] signers;
  readonly Object[] getSigners() {
    return signers;
  }
}
```

The `readonly` keyword ensures that the caller of `Class.getSigners` cannot modify the returned array, thus permitting the simple and efficient implementation of the method to remain in place without exposing the representation to undesired changes.[8]

An alternate solution to the `getSigners` problem, which was actually implemented in later versions of the JDK, is to return a copy of the array `signers` [41]. This works,

[8] The returned array is aliased by the `signers` field, so `Class` code can still change it even if external code cannot. The specification of `getSigners` does not state the desired semantics in this case, so this is an acceptable implementation. If a different semantics were desired, the method specification could note that the returned reference reflects changes to the signers of the `Class`; alternately, the method specification, or an external analysis, might require that the result is used before the next modification of `signers`.

but is expensive. For example, a file system may allow a client read-only access to its contents:

```
class FileSystem {
  private List<Inode> inodes;
  List<Inode> getInodes() {
    ··· //Unrealistic to copy
  }
}
```

Javari allows the programmer to avoid the high cost of copying `inodes` by writing the return type of the method as:

```
readonly List<readonly Inode> getInodes()
```

This return type prevents the `List` or any of its contents from being modified by the client. As with all parameterized classes, the client specifies the type argument, including whether it is read-only or not, independently of the parameterized typed.

In this case, the list returned is declared to be read-only and contain read-only elements, and, thus, a client of `getInodes` is unable to modify the list or its elements.

4.2 Cases from OIGJ

Figure 1 shows an implementation of `LinkedList` in OIGJ that follows closely the Sun's implementation but does not contain any additional bounds checking and supporting code that would prevent it from fitting on one page. 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 the creation of immutable lists (lines 7–21), and (iii) finally we dive into the complexities of inner classes and iterators (lines 23–47). Note that method guards [42] state that the method is only applicable if the type variable matches the bound stated by the guard (e.g. method `next` inside the `ListItr` can only be called if `ItrI` is a subtype of `Mutable`).

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.

Note that we assume that `O` refers to the current class instance owner and `I` or `ItrI` refer to the appropriate current immutability that is either class instance or method call specific.

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

```java
class Entry<O,I,E> {
  E element;
  Entry<O,I,E> next, prev;
}
class LinkedList<O,I,E> {
  Entry<This,I,E> header;
  <I extends Raw>? LinkedList() {
    this.header = new Entry<This,I,E>();
    header.next = header.prev = header;
  }
  <I extends Raw>? LinkedList(Collection<?,ReadOnly,E> c) {
    this(); this.addAll(c);
  }
  <I extends Raw>? void addAll(Collection<?,ReadOnly,E> c) {
    Entry<This,I,E> succ = this.header, pred = succ.prev;
    for (E e : c) {
      Entry<This,I,E> en=new Entry<This,I,E>();
      en.element=e; en.next=succ; en.prev=pred;
      pred.next = en; pred = en; }
    succ.prev = pred;
  }
  int size() {···}
  <ItrI extends ReadOnly> Iterator<O,ItrI,I,E> iterator() {
    return this.new ListItr<ItrI>();
  }
  void remove(Entry<This,Mutable,E> e) {
    e.prev.next = e.next;
    e.next.prev = e.prev;
  }
  class ListItr<ItrI> implements Iterator<O,ItrI,I,E> {
    Entry<This,I,E> current;
    <ItrI extends Raw>? ListItr() {
      this.current = LinkedList.this.header;
    }
    <ItrI extends Mutable>? E next() {
      this.current = this.current.next;
      return this.current.element;
    }
    <I extends Mutable>? void remove() {
      LinkedList.this.remove(this.current);
    }
} }
interface Iterator<O,ItrI,CollectionI,E> {
  boolean hasNext();
  <ItrI extends Mutable>? E next();
  <CollectionI extends Mutable>? void remove();
}
```

Fig. 1. `LinkedList<O,I,E>` in OIGJ

`this.header.next.next` is `Entry<This,I,E>`. Thus, the owner of all entries is the `this` object, i.e., the list. OIGJ provides *deep ownership* or *owners-as-dominators* guarantees as discussed in another chapter [43].

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. Phrased differently, `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 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`.

An object becomes *cooked* either when its new expression (construction) finishes executing or when its owner is *cooked*. The entries of the list (line 6) are `this`-owned. Indeed, the entries are mutated after their constructor has finished, but before their owner (list) is cooked, on lines 9, 19, and 20. 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, 15, and 33. In contrast, fields `next` and `prev` (which are not `this`-owned) do not have such a restriction, as can be seen on lines 27–28. Note that inner classes are treated differently, and are allowed to access the outer object's `this`-owned fields. This arguably gives a more flexible system, which for instance allows the iterator class in the example, but at the cost of less clear containment properties.

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 43–47, and used on lines 23 and 30. `ItrI` is the iterator's immutability, whereas `CollectionI` is intended to be the underlying collection's immutability (see on line 30 how the collection's immutability `I` is used in the place of `CollectionI`). Line 45 requires a mutable `ItrI` to call `next()`, and line 46 requires a mutable `CollectionI` to call `remove()`.

Inner class `ListItr` (lines 30–42) is the implementation of `Iterator` for list. Its full name is `LinkedList<O,I,E>.ListItr<ItrI>`, and on line 30 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 33), which would be illegal if their owner was different. For example, consider the types of `this` and `LinkedList.this` on line 33:

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

Because line 32 sets the bound of `ItrI` to be `Raw`, `this` can be mutated. By contrast, the bound of `I` is `ReadOnly`, so `LinkedList.this` cannot.

Finally, consider the creation of a new *inner* object on line 24 using $\underline{\texttt{this}}$`.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 23) and the guard on the constructor is `Raw` (line 32), which is legal because a `Raw` constructor can create both mutable and immutable objects.

4.3 Cases from Joe$_3$

This subsection presents a different take on how to specify read-only and immutability, in which the context where an object is used determines its mutability properties.

A short note on effects. Joe$_3$ employs a very simple effects system to specify what a class or method will mutate. The effects are specified in terms of contexts (or owners) inspired by Joe$_1$ [44]. In Joe$_3$ context parameters are decorated with modes which govern how objects belonging to a particular context may be treated.

Separating Mutability of List and its Contents. Figure 2 shows part of an implementation of a list class. The class is parameterized over permissions (`data` in this case) which specify that the list has privilege to reference objects in that context. The parameter `data` is decorated with the mode read-only (denoted '`-`'), indicating that the list will never cause write effects to objects owned by `data`. The owner of the list is called `owner` and is implicitly declared. The method `getFirst()` is annotated with `revoke owner`, which

```
class Link<data- strictlyoutside owner> {
  data:Object obj = null;
  owner:Link<data> next = null;
}

class List<data- strictlyoutside owner> {
  this:Link<data> first = null;
  void addFirst(data:Object obj) {
    this:Link<data> tmp = new this:Link<data>();
    tmp.obj = obj;
    tmp.next = this.first;
    this.first = tmp;
  }
  void filter(data:Object obj) {
    this:Link<data> tmp = this.first;
    if (tmp == null) return;
    while (tmp.next != null)
      if (tmp.next.obj == obj)
        tmp.next = tmp.next.next;
      else
        tmp = tmp.next;
    if (this.first != null && this.first.obj == obj)
      this.first = this.first.next;
  }
  data:Object getFirst() revoke owner { return this.first.obj; }
}
```

Fig. 2. Fragment of a list class. As the `data` owner parameter is declared read-only (via '`-`') in the class header, no method in `List` may modify an object owned by `data`. Observe that the syntactic overhead is minimal for an ownership types system.

means that the method will not modify the list or its transitive state. This means the same as if `owner-` and `this-` would have appeared in the class header. This allows the method to be called from objects where the list owner is read-only. Finally, `strictlyoutside` means that the `data` context must not be the same context as the `owner` of the list.

This list class can be instantiated in four different ways, depending on the access rights to the owners in the type held by the current context:

- both the list and its data objects are immutable, which only allows `getFirst()` to be invoked, and its resulting object is immutable;
- both are mutable, which imposes no restrictions;
- the list is mutable but the data objects are not, which imposes no additional restrictions, `getFirst()` returns a read-only reference; and
- the data objects are mutable, but the list not, which only allows `getFirst()` to be invoked, and the resulting object is mutable.

The last form is interesting and relies on the fact that it is known, courtesy of ownership types, that the data objects are not part of the representation of the list. Without this distinction one could easily create an example where a mutable alias can be returned from a read-only object.

4.4 Complementing Immutability

In this subsection we explain why blindly adding immutability support to any OO language with aliasing might not be such a good idea and how it can be addressed by careful application of various ownership techniques discussed in the rest of this book.

Boyland [45] criticizes existing proposals for handling read-only references on the following points:

1. Read-only references can be aliased, for example by capturing a method argument;
2. A read-only annotation does not express whether
   (a) the referenced *object* is immutable, so the referent is known to never change;
   (b) a read-only reference is unique and thus effectively immutable;
   (c) mutable aliases of a read-only reference can exist, which makes possible *observational exposure*, which occurs when changes to state are observed through a read-only reference.

Essentially, Boyland is criticizing reference immutability for not being object immutability. In some contexts, object immutability is more useful, and in other contexts, reference immutability is more useful. Furthermore, as we noted earlier, different contexts require other differences (such as representation immutability for reasoning about the memory system, and abstract immutability for reasoning about client code semantics). Boyland's criticisms can be addressed, for those contexts where object immutability is desired, by augmenting reference immutability with object immutability.

For example, Joe$_3$ addresses all of these problems. First, Joe$_3$ supports owner-polymorphic methods, which can express that a method does not capture one or more of its arguments. Second, owners are decorated with modes that govern how the objects owned by that owner will be treated in a particular context. Together with auxiliary constructs inherited from Joline [46], the modes can express immutability both in terms of 2.a) and 2.b), and read-only which permits the existence of mutable aliases (2.c). Moreover, Joe$_3$ supports fractional permissions [47] — converting a mutable unique reference into several immutable references for a certain context. This allows safe representation exposure without the risk for observational exposure.

OIGJ and Joe$_3$ allow both read-only references and immutable objects in the same language. This provides the safety desired by Boyland's second point, but also allows coding patterns which do rely on observing changes in an object. In order to support such flexibility it appears necessary to employ some kind of *alias control* mechanism, which in the cases of OIGJ and Joe$_3$ is ownership.

Ownership and Immutability. Ownership types [40, 48] impose a structure on the references between objects in the program heap. Languages with ownership, such as for instance Joline [46] and OGJ [49], 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 which cannot break encapsulation, even though the idea was originally sprung out of a proposal supporting that [30].

One possible application of ownership types is the ability to reason about read and write effects [50] which has complimentary goals to object immutability. Universes [9] is a Java language extension combining ownership and read-only references. Most ownership systems enforce that all reference chains to an owned object pass through the owner. Universes relaxes this requirement by enforcing this rule only for mutable references, i.e., read-only references may be shared without restriction.

Universes, OIGJ, and Joe$_3$ provide what we call context-based immutability. Here it is possible to create a writable list with writable elements and pass it to some other context where the elements are read-only. This other context may add elements to the list (or reorder them) but not mutate the elements, while the original creator of the list does not lose the right to mutate the elements.

A read-only reference to an object does not preclude the existence of mutable references to the same object elsewhere in the system. This allows observational exposure [45] — for good and evil. Object immutability imposes all restrictions of a read-only reference, but also guarantees that no aliases with write permission exist in the system. One simple way of creating an immutable object is to move a *unique* reference into a variable with immutable type [51, 12].

5 A Selection of Recent Immutability Proposals

We will now present in more detail the major proposals improving on the status quo in modern popular OO languages, i.e., `final` in Java and `const` in C++. Javari [6] was presented in 2004, adding read-only references. In 2007 Immutability Generic Java (IGJ) [10] was proposed, adding immutability and immutability parameterization. Joe$_3$ [12], proposed in 2008, uses ownership, external uniqueness — to support transition from mutable to immutable — and a trivial effects system to support immutability and context-based immutability. OIGJ [11], published in 2010, again employs ownership to achieve immutability, albeit a less strict variant of ownership allowing for more flexibility. OIGJ uses the existing Java generics machinery to encode ownership, which removes the need for an extra parameter clause on classes and methods. While we describe the proposals by the authors of this chapter in great detail, we also provide an overview of the other complementary proposals in the Section 5.5.

5.1 Javari

The Javari [6] programming language extends Java to allow programmers to specify and enforce reference immutability constraints. A read-only reference cannot be used to modify the object, including its transitive state, to which it refers.

Javari's type system differs from previous proposals (for Java, C++, and other languages) in a number of ways. First, it offers reference, not object, immutability. Second, Javari offers guarantees for the entire transitively reachable state of an object — the state of the object and all state reachable by following references through its (non-static) fields. A programmer may use the type system to support reasoning about either the representation state of an object or its abstract state; to support the latter, parts of a class can be marked as not part of its abstract state.

Third, Javari combines static and dynamic checking in a safe and expressive way. Dynamic checking is necessary only for programs that use immutability downcasts, but such downcasts can be convenient for interoperation with legacy code or to express facts that cannot be proved by the type system. Javari also offers parameterization over immutability.

Experience with over 160,000 lines of Javari code, including the Javari compiler itself, indicates that Javari is effective in practice in helping programmers to document their code, reason about it, and prevent and eliminate errors.

The language design issues include the following:

- Should Javari use new keywords (and possibly other syntax) to indicate reference immutability, or should it use Java's annotation mechanism[9]? (Or, should a prototype implementation use annotations, even if Javari itself should eventually use keywords?)
- The immutability downcast adds expressiveness to the Javari language, but it also adds implementation complexity and (potentially pervasive) run-time overhead. Is a language that lacks immutability downcasts practical?
- How can uses of reflection, serialization, and other Java constructs that are outside the scope of the Java type system be handled, particularly without adding run-time overhead?
- The existing Javari template mechanism is unsatisfactory. It is orthogonal to Java generics (even in places where it seems that generics should be a satisfactory solution).

Javari has presented a type system that is capable of expression, compile-time verification, and run-time checking of reference immutability constraints. Read-only references guarantee that the reference cannot be used to perform any modification of a (transitively) referred-to object. The type system should be generally applicable to object-oriented languages, but for concreteness we have presented it in the context of Javari, an extension to the full Java 5 language, including generic types, arrays, reflection, serialization, inner classes, exceptions, and other idiosyncrasies. Immutability polymorphism (templates) for methods are smoothly integrated into the language, reducing code duplication. Tschantz et al. [6] provided a set of formal type rules for a core calculus that models the Javari language and used it to prove type soundness for the Javari type system.

Javari provides a practical and effective combination of language features. For instance, we describe a type system for reference rather than object immutability. Reference immutability is useful in more circumstances, such as specifying interfaces, or objects that are only sometimes immutable. Furthermore, type-based analyses can run after type checking in order to make stronger guarantees (such as object immutability) or to enable verification or transformation. The system is statically type-safe, but optionally permits downcasts that transform compile-time checks into run-time checks for specific references, in the event that a programmer finds the type system too constraining. The language is backward compatible with Java and the Java Virtual Machine, and is interoperable with Java. Together with substantial experience with a prototype for

[9] `http://java.sun.com/j2se/1.5.0/docs/guide/language/annotations.html`

a closely related dialect [3], these design features provide evidence that the language design is effective and useful.

The Javari language presented is an evolutionary improvement of an earlier dialect [3], which we call "Javari2004". In what follows we highlight the main features of the Javari language and justify them in the context of a large user study done using the previous version of the language.

Distinguishing Assignability from Mutability. Javari2004's `mutable` keyword declares that a field is both assignable and mutable: there is no way to declare that a field is only assignable or only mutable. Javari's `assignable` and `mutable` keywords highlight the orthogonality of assignability and mutability, and increase the expressiveness of the language. See appendix of Javari paper for examples of the use of `assignable` and `mutable`.

Generic Types. Javari provides a detailed treatment of generic classes that smoothly integrates read-only references with them. Javari2004 does not supports generic classes, though the OOPSLA 2004 paper speculates about a macro expansion mechanism that is syntactically, but not semantically, similar to the way that Java 5 treats type parameters. Java 5 compiles type parameters via type erasure, but Javari2004 treated the mutability parameters (which appeared in the same list as the type parameters) via code duplication; this distinction complicates implementation, understanding, and use.

Arrays. As with generic classes, Javari permits programmers to independently specify the mutability of each level of an array. By contrast, Javari2004's specification states: "`readonly int[][]` and `readonly (readonly int[])[]` are equivalent," forbidding creation of a read-only array of mutable items.

Within current Javari one may make such as declaration as follows: `readonly Object[]`. One could also declare a mutable array of read-only objects: `/*mutable*/ (readonly Object)[]`.

Method Templates. Javari2004 integrated the syntax for templating a method over mutability with the syntax for Java 5's generic types.

For example, the following method signature is a templated method.

```
public <RO> RO List<RO Invariant> get(RO PptTopLevel ppt) RO;
```

The `<RO>` at the beginning of the signature specifies that `RO` is a type parameter.

Whether a parameter is intended to be a normal type parameter or a mutability type parameter must be inferred from its usage, greatly complicating a compiler (and the prototype Javari2004 implementation required distinct syntax to ease the compiler's task [52, 3]).

Furthermore, Javari2004 allows declaring an arbitrary number of mutability type parameters. Only a single mutability type parameter was deemed sufficient by the Javari designers, so Javari uses a much simpler mechanism (`romaybe`) for indicating a variable mutability. This new approach highlights the orthogonality of the Java 5's generic types and Javari's mutability polymorphism for methods. Furthermore, it does not require any run-time representation of the polymorphism. IGJ [10] and OIGJ [11] have both demonstrated that building on generic types provides a much nicer treatment for immutability parameterization completely removing the need for `romaybe`.

5.2 IGJ

Immutability Generic Java. (IGJ) is a language that supports class and object immutability and read-only references. Each object is either mutable or immutable, and each reference is `Immutable`, `Mutable`, or `ReadOnly`. Inspired by work that combines ownership and generics [49], the distinctions are expressed without changing Java's syntax by adding one new type parameter (at the beginning of the list of type parameters):

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

Line 3 shows object immutability in IGJ, and Line 10 shows read-only references.

Java prohibits changes to type arguments, such as in Line 10, to avoid a type loophole. More detailed discussion of the Java language and type system is given by Bracha et al. [53] and Igarashi et al. [54]. Line 10 is legal in IGJ, because IGJ allows covariant changes in the immutability parameter. IGJ even allows covariant changes in other type parameters if mutation is disallowed, e.g., `List<ReadOnly,Integer>` is a subtype of `List<ReadOnly,Number>`.

IGJ satisfies the following design principles:

Transitivity. More accurately, IGJ does not require transitivity. Rather, it provides a mechanism by which programmers can specify exactly where transitivity should be applied — and then that transitivity is enforced by the type system.
IGJ provides transitive (deep) immutability that protects the entire abstract state of an object. For example, an immutable graph contains an immutable set of immutable edges.
`C++` does not support such transitivity because its `const`-guarantee does not traverse pointers, i.e., a pointer in a `const` object can mutate its referent.
IGJ also permits excluding a field from the abstract state. For example, fields used for caching can be mutated even in an immutable object.

Static. IGJ has no runtime representation for immutability, such as an "immutability bit" that is checked before assignments or method calls. IGJ designers believe that testing at runtime whether an object is immutable [4] 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 runtime penalty. A similar approach was taken by Generic Java (GJ) [53] 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: every legal Java program is a legal IGJ program.

Polymorphism. IGJ abstracts over immutability without code duplication by using generics and a flexible subtype relation. 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. A small number of additional typing rules make IGJ more restrictive than Java. On the other hand, IGJ's subtyping rules are more relaxed, allowing covariant changes in a type-safe manner.
Phrased differently, IGJ uses rules which fit naturally into Java's design.

Most of the IGJ terminology was borrowed from Javari [6] such as assignable, readonly, mutable, and `this`-mutable. In Javari, `this`-mutable fields are mutable as lvalue and readonly as rvalue. Javari does not support object immutability, and its read-only references are more limited than that of IGJ because Javari has no `this`-mutable parameters, return types, or local variables. Javari's keyword `romaybe` is in essence a template over immutability. IGJ uses generics directly to achieve the same goal.

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 { mutable List<Object> list; }
```

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

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

which is syntactic sugar for

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

Thus, it is possible to insert only mutable elements to `list`, and retrieve only read-only elements. Such complexities, in IGJ designers point of view, make IGJ easier to use than Javari.

5.3 Joe$_3$

This section will present Joe$_3$ in a bit more detail. Joe$_3$ is a Java-like language with deep ownership, owner-polymorphic methods, external uniqueness, an effects (revocation) system and a simple mode system which decorates owners with permissions to indicate how references with the annotated owners may be used. The annotation of owners with modes is the main novelty in Joe$_3$. The modes indicate that a reference may be read or written (+), only read (-), or that the reference is immutable (*). Read and immutable annotations on an owner in the class header represent a promise that the code in the class body will not change objects owned by that owner. The key to preserving and respecting immutability and read-only in Joe$_3$ is a simple effects system, rooted in ownership types, and inspired by Clarke and Drossopoulou's Joe$_1$ [44]. Classes, and hence objects, have rights to read or modify objects belonging to certain owners; only a minor extension to the type system of Clarke and Wrigstad's Joline [55, 56] is required to ensure that these rights are not violated.

Classes are parameterized with owners related to each other by an inside/outside nesting relation. An owner is a permission to reference the representation of another object. Class headers have this form:

```
class List<data- outside owner> { ··· }
```

Each class has at least two owner parameters, `this` and `owner`, which represent the representation of the current object and the representation of the owner of the current object, respectively. In the example above, the `List` class has an additional permission to reference objects owned by `data`, which is nested outside `owner`. Types are formed by instantiating the owner parameters, `this:List<owner>`. An object with this type belongs to the representation of the current object and has the right to reference objects owned by `owner`. There are two nesting relations between owners, inside and outside. They exist in two forms each, one reflexive (`inside`/`outside`) and one non-reflexive (`strictly-inside`/`strictly-outside`). Thus, going back to our list example, a type `this:List<this>` denotes a list object belonging to the current representation, holding objects in the current representation.

Apart from ownership types, the key ingredients in Joe$_3$ are the following:

– (externally) unique types (written `unique[p]:Object`), a special *borrowing* construct for temporarily treating a unique type non-uniquely, and *owner casts* for converting unique references permanently into normal references.
– modes on owners — mutable '+', read-only '-', and immutable '*'. These appear on every owner parameter of a class and owner polymorphic methods, though not on types.
– an effects revocation clause on methods which states which owners will not be modified in a method. An object's default set of rights is derived from the modes on the owner parameters in the class declaration.

Annotating owners at the level of classes (that is, for all instances) rather than types (for each reference) is a trade-off. Rather than permitting distinctions to be made using modes on a per reference basis, Joe$_3$ admits only per class granularity. Some potential expressiveness is lost, though the syntax of types does not need to be extended. Nonetheless, the effects revocation clauses regain some expressiveness that per reference modes would give. Another virtue of using per class rather than per reference modes is that some covariance problems found in other proposals are avoided, as what you can do with a reference depends on the context and is not a property of the reference. The covariance problem, similar to Java generics, is essentially that immutability variance in the element parameter of a list makes it possible to mutate read-only elements.

```
class ListWriter<o+ outside owner, data- strictlyoutside o> {
  void mutateList(o:List<data> list) {
    list.addFirst(new data:Object());
  }
}
class ListReader<o- outside owner, data+ strictlyoutside o> {
  void mutateElements(o:List<data> list) {
    list.elementAt(0).mutate();
  }
}
class Example {
  void example() {
    this:List<world> list = new this:List<world>();
    this:ListWriter<this, world> w = new this:Writer<this, world>();
    this:ListReader<this, world> r = new this:Reader<this, world>();
    w.mutateList(list);
    r.mutateElements(list);
  }
}
```

Fig. 3. Different views of the same list can exist at the same time. `r` can modify the elements of `list` but not the `list` itself, `w` can modify the `list` object, but not the list's contents, and instances of `Example` can modify both the list and its contents.

Context-Based Read-Only. As shown in Figure 3, different clients of the list can have different views of the same list at the same time. The class `ListReader` does not have permission to mutate the list, but has no restrictions on mutating the list elements. Dually, the `ListWriter` class can mutate the list but not its elements.

As owner modes only reflect what a class is allowed to do to objects with a certain owner, `ListWriter` can add data objects to the list that are read-only to itself and the list, but writable by `Example` and `ListReader`. This is a powerful and flexible idea. For example, `Example` can pass the list to `ListWriter` to filter out certain objects in the list. `ListWriter` can then consume or change the list, or copy its contents to another list, *but not modify them*. `ListWriter` can then return the list to `Example`, without `Example` losing its right to modify the objects obtained from the returned list. This is similar to the context-based read-only in Universes-based systems [57, 58]. In contrast, however, Joe$_3$ does not allow representation exposure via read-only references.

Immutable Object Initialization. Immutable objects need to be mutated in their construction phase. Unless caution is taken the constructor might leak a reference to `this` (by passing `this` to a method) or mutate other immutable objects of the same class. The standard solution to this problem in related proposals is to limit the construction phase to the constructor [10, 11, 59]. Continuing initialization by calling auxiliary methods *after* the constructor returns is simply not possible. Joe$_3$, on the other hand, permits *staged construction*, as demonstrated in Figure 4. In this example a client uses a factory to create an immutable list. The factory creates a unique list and populates it. The list is then destructively read and returned to the caller as an immutable. Interestingly enough, if

```
class Client<p* outside owner, data+ strictlyoutside p> {
  void method() {
    this:Factory<p, data> f = new this:Factory<p, data>();
    p:List<data> immutable = f.createList();
  }
}
class Factory<p* inside world, data+ strictlyoutside p> {
  p:List<data> createList() {
    unique[p]:List<data> list = new p:List<data>();
    borrow list as temp+ l in { // 2nd stage of construct.
      l.add(new data:Object());
    }
    return list--; // unique reference returned
  }
}
```

Fig. 4. Staged construction of an immutable list

an inference mechanism were employed to remove trivial uses of borrowing, such as the one in the figure, then this example could be done without the extra baggage.

Fractional Permissions. Using uniqueness and Joline's borrowing statement, Joe$_3$ can encode a variant of Boyland's Fractional Permissions [47], where a mutable reference is turned into an immutable reference for a limited time, after which it can be reestablished as a mutable reference with no residual aliasing. This is described in more detail with an example in Section 6.3.

5.4 OIGJ

This section presents the OIGJ language extension that expresses both ownership and immutability information.

OIGJ introduces two new type parameters to each type, called the *owner parameter* and the *immutability parameter*. For simplicity of presentation, 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. The owner and immutability parameters form two separate hierarchies, which are shown in Figure 5. These parameters cannot be extended, and they have no subtype relation with any other types. The subtyping relation is denoted by $\preceq$, e.g., `Mutable` $\preceq$ `ReadOnly`. Subtyping is invariant in the owner parameter and covariant in the immutability parameter.

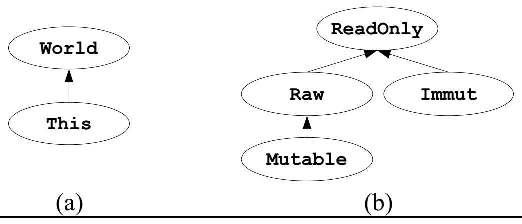

Fig. 5. The type hierarchy of (a) ownership and (b) immutability parameters. `World` means the entire world can access the object, whereas `This` means that `this` owns the object and no one else can access it. A `Immut`/`Mutable` reference points to an immutable/mutable object. A `ReadOnly` reference 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.

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) [42], 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. Figure 6 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 3–10 show different kinds of immutability in OIGJ: immutable, mutable, and readonly. A read-only and an immutable reference may seem similar because neither can be used to mutate the referent. However, line 10 shows the difference between the two: a read-only reference may point to a mutable object. Phrased differently, a read-only reference may not mutate its referent, though the referent may be changed via an aliasing mutable reference.

Java's type arguments are invariant (neither covariant nor contravariant), to avoid a type loophole [54], so line 10 is illegal in Java. Line 10 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.

```
 1 class Foo<O extends World,I extends ReadOnly> {
 2   // An immutable reference to an immutable date.
 3   Date<O,Immut> imD = new Date<O,Immut>();
 4   // A mutable reference to a mutable date.
 5   Date<O,Mutable> mutD = new Date<O,Mutable>();
 6   // A read−only reference to any date.
 7   // Both roD and imD cannot mutate their referent,
 8   // however the referent of roD might be mutated by an alias,
 9   // whereas the referent of imD is immutable.
10   Date<O,ReadOnly> roD = ··· ? imD : mutD;
11   // A date with the same owner and immutability as this.
12   Date<O,I> sameD;
13   // A date owned by this; it cannot leak.
14   Date<This,I> ownedD;
15   // Anyone can access this date.
16   Date<World,I> publicD;
17   // Can be called on any receiver; cannot mutate this.
18   // The method guard ‘‘<···>?’’ is part of cJ’s syntax˜[42].
19   <I extends ReadOnly>? int readonlyMethod() {···}
20   // Can be called only on mutable receivers; can mutate this.
21   <I extends Mutable>? void mutatingMethod() {···}
22   // Constructor that can create (im)mutable objects.
23   <I extends Raw>? Foo(Date<O,I> d) {
24     this.sameD = d;
25     this.ownedD = new Date<This,I>();
26     // Illegal, because sameD came from the outside.
27     // this.sameD.setTime(···);
28     // OK, because Raw is transitive for owned fields.
29     this.ownedD.setTime(···);
30   } }
```

Fig. 6. An example of OIGJ syntax

The immutability of `sameD` (line 12) 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 12–16 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 12–16 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., $O($ownedD$) = $ This. Similarly, function $I(\ldots)$ returns the immutability parameter; e.g., $I($ownedD$) = $ I. We say that an object o is *this-owned* (i.e., owned by this) if $O($o$) = $ 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) = $ This then $\theta(o) = $ this, if $O(o) = $ O then $\theta(o) = \theta($this$)$, and if $O(o) = $ World then $\theta(o) = $ 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, OIGJ provides the following *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) $o \preceq_\theta$ World. By part (iii), if $\theta(o) = $ World then anyone can point to o. On lines 12–16, we see that this can point to ownedD, sameD, publicD, whose owner parameters are This, O, World, and whose owners are this, $\theta($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<$\underline{\text{This}}$,I,Date<$\underline{\text{O}}$,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 19 and 21 of Figure 6 show a read-only and a mutating method. These methods are *guarded* with <...>?. Conditional Java (cJ) [42] 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 <...>$\underline{?}$. OIGJ paper uses cJ for convenience. However, the OIGJ implementation uses type annotations [60] 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 21 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 used *declaration annotations* to denote the immutability of `this`. In this chapter, OIGJ uses cJ to reduce the number of typing rules and handle inner classes more flexibly.[10] 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.

To summarize, on lines 19–23 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 19 can be removed.

Constructor example. The constructor on line 23 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.

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);
}
```

OIGJ provides the following *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 23–30 shows this difference between the assignments to `sameD` (line 24) and to `ownedD` (line 25): `sameD` can come from the outside world, whereas `ownedD` must be created inside `this`. Thus, `sameD` cannot be further mutated (line 27) whereas `ownedD` can be mutated (line 29) 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 27 is illegal; however, the receiver of `ownedD.setTime(...)` is `this`-owned, and therefore the call on line 29 is legal.

5.5 Other Immutability Proposals

JAC Similarly to the proposals in this chapter, JAC [7] has a `readonly` keyword indicating transitive immutability, an implicit type `readonly T` for every class and

[10] The OIGJ 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.

interface T defined in the program, and a mutable keyword. However, the other aspects of the two languages' syntax and semantics are quite different. For example, JAC provides a number of additional features, such as a larger access right hierarchy (readnothing $<$ readimmutable $<$ readonly $<$ writeable) and additional keywords (such as nontransferrable) that address other concerns than immutability. The JAC authors propose implementing JAC by source rewriting, creating a new type readonly T that has as methods all methods of T that are declared with the keyword readonly following the parameter list (and then compiling the result with an ordinary Java compiler). However, the return type of any such method is readonly. For example, if class Person has a method public Address getAddress() readonly, then readonly Person has method public readonly Address getAddress() readonly. In other words, the return type of a method call depends on the type of the receiver expression and may be a supertype of the declared type, which violates Java's typing rules. Additionally, JAC is either unsound for, or does not address, arrays of readonly objects, casts, exceptions, inner classes, and subtyping. JAC readonly methods may not change any static field of any class. The JAC paper suggests that readonly types can be supplied as type variables for generic classes without change to the GJ proposal, but provides no details. By contrast to JAC, in Javari the return type of a method does not depend on whether it is called through a read-only reference or a non-read-only one. Both IGJ and OIGJ provide a thorough treatment of the generic types and immutability bringing the JAC generics proposal to its logical conclusion.

The above comments also explain why use of read-only interfaces in Java is not satisfactory for enforcing read-only references. A programmer could define, for every class C, an interface RO_C that declares the read-only methods and that achieves transitivity through changing methods that returned (say) B to return RO_B. Use of RO_C could then replace uses of Javari's readonly C. This is similar to JAC's approach and shares similar problems. For instance, to permit casting, C would need to implement RO_C, but some method return and argument types are incompatible. Furthermore, this approach does not allow read-only versions of arrays or even Object, since RO_Object would need to be implemented by Object. It also forces information about a class to be maintained in two separate files, and it does not address run-time checking of potentially unsafe operations or how to handle various other Java constructs. Javari sidesteps these fundamental problems by extending the Java type system rather than attempting to work within it.

Modes for Read-Only. Skoglund and Wrigstad [4] take a different attitude toward immutability than other work: "In our point of [view], a read-only method should only protect its enclosing object's transitive state when invoked on a read reference but not necessarily when invoked on a write reference." A read (read-only) method may behave as a write (non-read-only) method when invoked via a write reference; a caseModeOf construct permits run-time checking of reference writeability, and arbitrary code may appear on the two branches. This suggests that while it can be proved that read references are never modified, it is not possible to prove whether a method may modify its argument. In addition to read and write references, the system provides context and any references that behave differently depending on whether a method is invoked on a read or write context. Compared to this work and JAC, Javari's, IGJ's and OIGJ's type parameterization gives a less ad hoc and more disciplined way to specify families of declarations.

Immutability Specification. Pechtchanski and Sarkar [5] provide a framework for immutability specification along three dimensions: lifetime, reachability, and context. The lifetime is always the full scope of a reference, which is either the complete dynamic lifetime of an object or, for parameter annotations, the duration of a method call. The reachability is either shallow or deep. The context is whether immutability applies in just one method or in all methods. The authors provide 5 instantiations of the framework, and they show that immutability constraints enable optimizations that can speed up some benchmarks by 5–10%.

Even if all methods of a class are state preserving, the resulting instances might not be immutable, because a mutable `this`object could escape the constructor and its fields can be mutated directly, for instance, if the constructor stores all created objects in a static set. The proposals in this chapter permit both of the lifetimes and supplies deep reachability, which complements the shallow reachability provided by Java's `final` keyword.

Capabilities. Capabilities for sharing [8] are intended to generalize various other proposals for immutability and uniqueness. When a new object is allocated, the initial pointer has 7 access rights: read, write, identity (permitting address comparisons), exclusive read, exclusive write, exclusive identity, and ownership (giving the capability to assert rights). Each (pointer) variable has some subset of the rights. These capabilities give an approximation and simplification of many other annotation-based approaches.

Why not add read-only? Boyland [61] explains "Why we should not add `readonly` to Java (yet)" and concludes that readonly does not address observational exposure, i.e., modifications on one side of an abstraction boundary that are observable on the other side. IGJ's immutable objects address such exposure because their state cannot change, e.g., an immutable address in a person object can be safely shared among many person objects. Sometimes it is impossible to avoid observational exposure, e.g., when a container changes and iterators to the inside of the container exists. Java designed its iterator classes to be *fail-fast*, i.e., the iterator will fail if the collection is mutated (which cannot happen in immutable collections).

Boyland's second criticism was that the *transitivity* principle (see end of Section 3) should be selectively applied by the designer, because, "the elements in the container are not notionally part of the container" [61]. In Joe$_3$, IGJ, and OIGJ, a programmer can solve this problem by using a different immutability for the container and its elements.

Effects. Effect systems [62–64] specify what state (in terms of regions or of individual variables) can be read and modified by a procedure; they can be viewed as labeling (procedure) types with additional information, which the type rules then manipulate. Type systems for immutability can be viewed as a form of effect system. The proposals in this chapter are finer-grained than typical effect systems, operate over references rather than values, and consider all state reachable from a reference.

Universes. Universe Types [65] were the first ownership type system to provide support for read-only references by introducing "owners as modifiers" discipline. Here, if an object does not have a right to access an object because of ownership restrictions, it is still allowed a read-only reference to such object. This greatly improved the expressiveness of the language albeit weakened the ownership guarantees - as a result both OIGJ and Joe$_3$ tried to mitigate these issues by providing a more granular interaction between immutability and ownership properties in a language. Finally, the functional methods of Universes [65] are pure methods that are not allowed to modify anything (as opposed to merely not being allowed to modify the receiver object).

Immutable Objects. Immutable Objects for a Java-like Language (**IOJ**) [66] 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 [66] later added flexible initialization of immutable objects [59], 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.)

Immutability Inference. Porat et al. [67] provide a type inference that determines (deep) immutability of fields and classes. (Foster et al. [37] provide a type inference for C's (non-transitive) `const`.) A field is defined to be immutable if its value never changes after initialization and the object it refers to, if any, is immutable. An object is defined to be immutable if all of its fields are immutable. A class is immutable if all its instances are. The analysis is context-insensitive in that if a type is mutable, then all the objects that contain elements of that type are mutable. Libraries are neither annotated nor analyzed: every virtual method invocation (even `equals`) is assumed to be able to modify any field. The paper discusses only class (static) variables, not member variables. The technique does not apply to method parameters or local variables, and it focuses on object rather than reference immutability, as in Javari. An experiment indicted that 60% of static fields in the Java 2 JDK runtime library are immutable.

Constructing Immutable Objects. Non-null types [68–71] 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, express the staged initialization paradigm in which the construction of an object continues after its constructor finishes. OIGJ [11] however addressed most of these IGJ shortcomings.

Frozen Objects. Leino et al. [72] 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. Frozen Objects show how flexible the initialization stage can potentially be in the presence of ownership and immutability, while, for example, OIGJ shows how much flexibility can be achieved while staying at the type checking level.

Other Proposals. Huang et al. [42] propose an extension of Java (called cJ) that allows methods to be provided only under some static subtyping condition. For instance, a cJ generic class, `Date<I>`, can define

```
<I extends Mutable>? void setDate(···)
```

which will be provided only when the type provided for parameter `I`is a subtype of `Mutable`. Designing IGJ on top of cJ would make METHOD-INVOCATION RULEredundant, at the cost of replacing IGJ's method annotations with cJ's conditional method syntax. The rest of IGJ's typing rules will remain the same.

Finally, IGJ uses the type system to check immutability statically. Controlling immutability at runtime (for example using assertions or Eiffel-like contractual obligations) falls outside the scope of this chapter.

5.6 Summary of Immutability Work

Figure 7 summarizes several proposals and their supported features. The systems included in the table represent the state of the art of read-only and immutable. Except Joe$_3$, the table includes (in order) SafeJava [51], Universes [57, 73–75], Jimuva [76], Javari [77], IGJ [78], JAC [79], ModeJava [80] and Effective Ownership [81]. We now discuss the different features in the table.

Joe$_3$ and SafeJava support staged construction of immutables.

Boyland suggests that copying rights may lead to observational exposure and proposes that the rights instead be split. Only the one with a complete set of rights may modify an object. SafeJava does not support borrowing to immutables and hence cannot model fractional permissions. It is unclear how allowing borrowing to immutables in SafeJava would affect the system, especially in the presence of back doors that break encapsulation.

To be able to retrieve writable objects from a read-only list, the elements in the list cannot be part of the list's representation. Joe$_3$, Universes, Jimuva and SafeJava can express this through ownership types. Only OIGJ and Joe$_3$ systems, thanks to owner nesting information, allow two non-sibling lists to share mutable data elements. Javari and IGJ allow this through ad-hoc mutable fields which can circumvent read-only if an object stores a reference to itself in a mutable field.

The alias modes proposed by Noble et al. [82]. Here we only describe how the modes have been interpreted for the purpose of Figure 7. The *rep* mode denotes a reference belonging to an object's representation and so should not appear in its interface. A defensive interpretation of *arg* is that all systems that have object or class immutability partially support *arg*, but only OIGJ and Joe$_3$ systems support *parts* of an object being immutable. The *free* mode, interpreted as being equal to uniqueness, is supported by

Feature	*OIGJ*	*Joe*$_3$	*SafeJava*	*Universes*	*Jimuva*	*Javari*	*IGJ*	*ModeJava*	*JAC*	*EO*
Expressiveness										
Staged const.	$\times^8$	$\surd$	$\surd$	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$
Fract. perm.	$\times$	$\surd$	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$
Non-rep fields	$\surd$	$\surd$	$\surd^1$	$\surd^1$	$\surd^1$	$\times^2$	$\times^2$	$\times$	$\times$	$\times$
Flexible Alias Protection Modes										
arg	$\surd$	$\surd$	$\times^3$	$\times$	$\times^3$	$\times^3$	$\times^3$	$\times^3$	$\times$	$\times$
rep	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\times$	$\times$	$\times$	$\times$	$\surd$
free	$\times$	$\surd$	$\surd$	$\surd$	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$
val [4]	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$
var	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$
Immutability										
Class	$\surd$	$\surd$	$\times$	$\times$	$\surd$	$\surd$	$\surd$	$\times^5$	$\times^5$	$\times$
Object	$\surd$	$\surd$	$\surd$	$\times$	$\surd$	$\times$	$\surd$	$\times$	$\times$	$\times$
Context-based	$\surd$	$\surd$	$\times$	$\surd$	$\times$	$\times^6$	$\times^6$	$\times^6$	$\times$	$\times$
Read-only ref.	$\surd$	$\surd$	$\times$	$\surd$	$\times$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$
Shallow Immutability	$\surd$	$\surd$	$\times$	$\surd$	$\times$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$
Deep Immutability	$\surd$	$\surd$	$\times$	$\times$	$\times$	$\surd$	$\surd$	$\times$	$\times$	$\times$
Abstract Value Immutability	$\surd$	$\surd$	$\times$	$\surd$	$\times$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$
Concrete Representation Immutability	$\surd$	$\surd$	$\times$	$\surd$	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$
Confinement and Alias Control										
OT	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\times$	$\times$	$\times$	$\times$	$\surd$
OP meths	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\times$	$\times$	$\times$	$\times$	$\times$
OAM	$\times$	$\times$	$\times$	$\surd$	$\times$	$\times$	$\times$	$\times$	$\times$	$\surd$
OAD	$\times^7$	$\surd$	$\times^7$	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$
Uniqueness	$\times$	$\surd$	$\surd$	$\surd$	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$

Fig. 7. Brief overview of related work. OT=ownership types, OP=owner polymorphic, OAM=owner-as-modifier, OAD=owner-as-dominator, EO=Effective Ownership. [1]) not as powerful as there is no owner nesting; two non sibling lists cannot share *mutable* data elements; [2]) mutable fields can be used to store a reference to `this` and break read-only; [3]) no modes on owners, and hence no immutable parts of objects; [4]) none of the systems deal with value semantics for complex objects; [5]) if all methods of a class are read-only the class is effectively immutable; [6]) limited notion of contexts via `this`-mutability; [7]) allows breaking of owners-as-dominators with inner classes. [8]) `Raw` constructors can construct both mutable and immutable objects, but once the constructor returns immutability is fixed.

Joe$_3$ and SafeJava. No system handles *val*ue semantics except for primitive types. The *var* aliasing mode expresses non-*rep* references which may be aliased and changed freely as long as they do not interfere with the other modes, for example, in assignments.

6 Discussion

We have shown so far how immutability support can be extended from non-transitive read-only references (e.g. `const` or `final`) to transitive read-only references (Javari) and further to object immutability (OIGJ and Joe$_3$) with ownership-like features. A number of interesting observations deserve further discussion, including initialization of immutable objects and covariant subtyping in the presence of immutability information as presented below. We also discuss type states and unique references with fractional permissions as further extensions that can complement immutability.

6.1 Covariant Subtyping

Covariant subtyping allows type arguments to covariantly change in a type-safe manner. Variant parametric types [83] attach a variance annotation to a type argument, e.g., `Vector<+Number>` (for covariant typing) or `Vector<-Number>` (for contravariant typing).

Its subtype relation contains this chain:

`Vector<Integer>` $\preceq$ `Vector<+Integer>` $\preceq$ `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>`
$\preceq$ `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>`. A variant parametric type system prohibits such an invocation. Java permits such an invocation, but the only value of type `? extends Number` is `null`.

IGJ also contains a similar chain:

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

The restriction on method calls in IGJ is based on user-chosen *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. IGJ is still type-safe because of the fact that `isIn` is readonly and the restriction on method overriding [10].

6.2 Typestates for Objects

In a typestate 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 its aliases. Typestates for objects [84] describes a system called Fugue that 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

```
 1 class Client {
 2   <p* inside world> void m1(p:Object obj) {
 3     obj.mutate(); // Error
 4     obj.toString(); // Ok
 5     // assign to field is not possible
 6   }
 7   <p- inside world> void m2(p:Object obj) {
 8     obj.mutate(); // Error
 9     obj.toString(); // Ok
10   }
11 }
12 class Fractional<o+ outside owner> {
13   unique[this]:Object obj = new this:Object();
14   void example(o:Client c) {
15     borrow obj as p*:tmp in {
16       c.m1(tmp);
17       c.m2(tmp);
18     }
19   }
20 }
```

Fig. 8. Fractional permissions using borrowing and unique references

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 [84] can handle arbitrary aliases that occur in real programs, e.g., `this` references that escape the constructor.

6.3 Fractional Permissions

The example in Figure 3 shows that a read-only reference to an object does not preclude the existence of mutable references to the same object elsewhere in the system. This allows observational exposure — for good and evil. The immutability annotation '`*`' imposes all the restrictions a read-only type has, but it also guarantees that no aliases with write permission exist in the system. Joe$_3$'s simple way of creating an immutable object is to move a *unique* reference into a variable with immutable type, just as in SafeJava [51]. This allows Joe$_3$ to encode fractional permissions using a borrowing construct and do staged construction of immutables. The example in Figure 8 shows an implementation of Fractional Permissions. Joline's borrowing construct [46] is employed to *temporarily* move a mutable unique reference into an immutable variable (line 15), freely alias the reference (while preserving read-only) (lines 16 and 17), and then implicitly move the reference back into the unique variable again and make it mutable. This is essentially Boyland's Fractional Permissions [47]. Both owner-polymorphic methods and borrowing blocks guarantee not to capture the reference. A borrowed reference can be aliased any number of times in any context to which it has been exported, without the need to keep track of "split permissions" [47] as it is guaranteed that all permissions to alias the pointer are invalidated when the borrowing block exits. The price of this convenience is that the conversion from mutable to immutable and back again must be done in the same place.

Interestingly, `m1` and `m2` are equally safe to call from `example`. Both methods have revoked their right to cause write effects to objects owned by `p`, indicated by the `*` and `-` annotations on `p`, respectively. The difference between the two methods is that the first method knows that `obj` will not change under foot (making it safe to, for example, use `obj` as a key in a hash table), whereas the second method cannot make such an assumption.

7 Conclusion

In this chapter we have given a flavor of the variety of work on different kinds of immutability proposed for the modern object-oriented languages where aliasing plays an important role. We refer the reader to the respective papers for more information about each system described in this chapter and welcome any feedback.

References

1. Hogg, J., Lea, D., Wills, A., de Champeaux, D., Holt, R.: The Geneva Convention on the Treatment of Object Aliasing. In: Clarke, D., Noble, J., Wrigstad, T. (eds.) Aliasing in Object-Oriented Programming. LNCS, vol. 7850, pp. 7–14. Springer, Heidelberg (2013)
2. Clarke, D., Noble, J., Wrigstad, T. (eds.): Aliasing in Object-Oriented Programming. LNCS, vol. 7850. Springer, Heidelberg (2013)
3. Birka, A., Ernst, M.D.: A practical type system and language for reference immutability. In: OOPSLA, pp. 35–49. ACM Press, New York (2004)
4. Skoglund, M., Wrigstad, T.: A mode system for read-only references in Java. In: FTfJP. Springer, Heidelberg (2001)
5. Pechtchanski, I., Sarkar, V.: Immutability specification and its applications. In: Java Grande, pp. 202–211. ACM Press, Seattle (2002)
6. Tschantz, M.S., Ernst, M.D.: Javari: Adding reference immutability to Java. In: OOPSLA, pp. 211–230. ACM Press, New York (2005)
7. Kniesel, G., Theisen, D.: JAC — access right based encapsulation for Java. Software: Practice and Experience 31(6), 555–576 (2001)
8. Boyland, J., Noble, J., Retert, W.: Capabilities for Sharing: A Generalisation of Uniqueness and Read-Only. In: Lindskov Knudsen, J. (ed.) ECOOP 2001. LNCS, vol. 2072, pp. 2–27. Springer, Heidelberg (2001)
9. Dietl, W., Müller, P.: Universes: Lightweight ownership for JML. Journal of Object Technology (JOT) 4(8), 5–32 (2005)
10. Zibin, Y., Potanin, A., Artzi, S., Kiezun, A., Ernst, M.D.: Object and reference immutability using Java generics. In: Foundations of Software Engineering (2007)
11. Zibin, Y., Potanin, A., Li, P., Ali, M., Ernst, M.D.: Ownership and immutability in generic java. In: OOPSLA, pp. 598–617. ACM Press (2010)
12. Östlund, J., Wrigstad, T., Clarke, D., Åkerblom, B.: Ownership, Uniqueness and Immutability. In: Paige, R.F., Meyer, B. (eds.) TOOLS EUROPE 2008. LNBIP, vol. 11, pp. 178–197. Springer, Heidelberg (2008)
13. Burdy, L., Cheon, Y., Cok, D., Ernst, M.D., Kiniry, J., Leavens, G.T., Leino, K.R.M., Poll, E.: An overview of JML tools and applications. STTT 7(3), 212–232 (2005)

14. Tkachuk, O., Dwyer, M.B.: Adapting side effects analysis for modular program model checking. In: ESEC/FSE, pp. 188–197. ACM Press, New York (2003)
15. Clausen, L.R.: A Java bytecode optimizer using side-effect analysis. Concurrency: Practice and Experience 9(11), 1031–1045 (1997)
16. Sălcianu, A.: Pointer analysis for Java programs: Novel techniques and applications. PhD thesis, MIT Dept. of EECS (September 2006)
17. Fowler, M.: Refactoring: Improving the Design of Existing Code. Addison-Wesley, New York (2000)
18. Artzi, S., Ernst, M.D., Kieżun, A., Pacheco, C., Perkins, J.H.: Finding the needles in the haystack: Generating legal test inputs for object-oriented programs. In: M-TOOS. ACM Press, Portland (2006)
19. Mariani, L., Pezzè, M.: Behavior capture and test: Automated analysis of component integration. In: ICECCS, pp. 292–301. IEEE, Tokyo (2005)
20. Xie, T.: Augmenting Automatically Generated Unit-Test Suites with Regression Oracle Checking. In: Thomas, D. (ed.) ECOOP 2006. LNCS, vol. 4067, pp. 380–403. Springer, Heidelberg (2006)
21. Ernst, M.D., Cockrell, J., Griswold, W.G., Notkin, D.: Dynamically discovering likely program invariants to support program evolution. IEEE TSE 27(2), 99–123 (2001)
22. Dallmeier, V., Lindig, C., Wasylkowski, A., Zeller, A.: Mining object behavior with ADABU. In: WODA, pp. 17–24. ACM Press, New York (2006)
23. Dolado, J.J., Harman, M., Otero, M.C., Hu, L.: An empirical investigation of the influence of a type of side effects on program comprehension. IEEE TSE 29(7), 665–670 (2003)
24. Gosling, J., Joy, B., Steele, G., Bracha, G.: The Java Language Specification, 3rd edn. AW (2005)
25. Boyapati, C.: SafeJava: A Unified Type System for Safe Programming. PhD thesis, MIT Dept. of EECS (February 2004)
26. Sălcianu, A., Rinard, M.: Purity and Side Effect Analysis for Java Programs. In: Cousot, R. (ed.) VMCAI 2005. LNCS, vol. 3385, pp. 199–215. Springer, Heidelberg (2005)
27. Rountev, A.: Precise identification of side-effect-free methods in Java. In: Proceedings of ICSM, pp. 82–91. IEEE Computer Society (2004)
28. Landi, W., Ryder, B.G., Zhang, S.: Interprocedural side effect analysis with pointer aliasing. In: Proceedings of PLDI, pp. 56–67 (1993)
29. Pearce, D.J.: JPure: A Modular Purity System for Java. In: Knoop, J. (ed.) CC 2011. LNCS, vol. 6601, pp. 104–123. Springer, Heidelberg (2011)
30. Noble, J., Vitek, J., Potter, J.: Flexible Alias Protection. In: Jul, E. (ed.) ECOOP 1998. LNCS, vol. 1445, pp. 158–185. Springer, Heidelberg (1998)
31. Kernighan, B.W., Ritchie, D.M.: The C Programming Language, 2nd edn. Software Series. Prentice Hall, Englewood Cliffs (1988)
32. Stroustrup, B.: The C++ Programming Language. Addison-Wesley, Boston (2000)
33. Meyers, S.: Effective C++, 2nd edn. Addison-Wesley (1997)
34. Morris, J.H.: Sniggering type checker experiment. Experiment at Xerox PARC (1978); Personal communication (May 2004)
35. Gannon, J.D.: An experimental evaluation of data type conventions. Communications of the ACM 20(8), 584–595 (1977)
36. Prechelt, L., Tichy, W.F.: A controlled experiment to assess the benefits of procedure argument type checking. IEEE TSE 24(4), 302–312 (1998)
37. Foster, J.S., Fähndrich, M., Aiken, A.: A theory of type qualifiers. In: PLDI, pp. 192–203 (June 1999)
38. Milner, R., Tofte, M., Harper, R.: The Definition of Standard ML. MIT Press (1990)
39. Leroy, X.: The Objective Caml system, release 3.07 (September 29, 2003) with Doligez, D., Garrigue, J., Rémy, D., Vouillon, J.

40. Clarke, D., Potter, J., Noble, J.: Ownership Types for Flexible Alias Protection. In: OOPSLA, pp. 48–64. ACM Press, Vancouver (1998)
41. Bloch, J.: Effective Java Programming Language Guide. Addison Wesley, Boston (2001)
42. Huang, S.S., Zook, D., Smaragdakis, Y.: cJ: Enhancing Java with safe type conditions. In: AOSD, pp. 185–198. ACM Press, New York (2007)
43. Clarke, D., Östlund, J., Sergey, I., Wrigstad, T.: Ownership Types: A Survey. In: Clarke, D., Noble, J., Wrigstad, T. (eds.) Aliasing in Object-Oriented Programming. LNCS, vol. 7850, pp. 15–58. Springer, Heidelberg (2013)
44. Clarke, D., Drossopoulou, S.: Ownership, Encapsulation, and the Disjointness of Type and Effect. In: OOPSLA, pp. 292–310. ACM Press, Seattle (2002)
45. Boyland, J.: Why we should not add readonly to Java (yet). Journal of Object Technology (2006); Special issue: ECOOP 2005 Workshop FTfJP
46. Wrigstad, T.: Ownership-Based Alias Management. PhD thesis, Royal Institute of Technology, Sweden (May 2006)
47. Boyland, J.: Checking Interference with Fractional Permissions. In: Cousot, R. (ed.) SAS 2003. LNCS, vol. 2694, pp. 55–72. Springer, Heidelberg (2003)
48. Clarke, D.: Object Ownership and Containment. PhD thesis, School of Computer Science and Engineering, University of New South Wales, Sydney, Australia (2001)
49. Potanin, A., Noble, J., Clarke, D., Biddle, R.: Generic ownership for generic Java. In: OOPSLA, pp. 311–324. ACM Press, New York (2006)
50. Clarke, D., Drossopoulou, S.: Ownership, encapsulation and the disjointness of type and effect. In: OOPSLA, pp. 292–310. ACM Press, New York (2002)
51. Boyapati, C.: SafeJava: A Unified Type System for Safe Programming. PhD thesis, Electrical Engineering and Computer Science, MIT (February 2004)
52. Birka, A.: Compiler-enforced immutability for the Java language. Technical Report MIT-LCS-TR-908, MIT Lab for Computer Science (June 2003); Revision of Master's thesis
53. Bracha, G., Odersky, M., Stoutamire, D., Wadler, P.: Making the future safe for the past: Adding genericity to the Java programming language. In: OOPSLA, pp. 183–200. ACM Press, New York (1998)
54. Igarashi, A., Pierce, B.C., Wadler, P.: Featherweight Java: a minimal core calculus for Java and GJ. ACM Transactions on Programming Languages and Systems (TOPLAS) 23(3), 396–450 (2001)
55. Clarke, D., Wrigstad, T.: External Uniqueness is Unique Enough. In: Cardelli, L. (ed.) ECOOP 2003. LNCS, vol. 2743, pp. 176–241. Springer, Heidelberg (2003)
56. Wrigstad, T.: Ownership-Based Alias Management. PhD thesis, Royal Institute of Technology, Kista, Stockholm (May 2006)
57. Müller, P., Poetzsch-Heffter, A.: Universes: A type system for controlling representation exposure. Technical report, Fernuniversität Hagen (1999)
58. Müller, P.: Modular Specification and Verification of Object-Oriented Programs. PhD thesis, FernUniversität Hagen (2001)
59. Haack, C., Poll, E.: Type-Based Object Immutability with Flexible Initialization. In: Drossopoulou, S. (ed.) ECOOP 2009. LNCS, vol. 5653, pp. 520–545. Springer, Heidelberg (2009)
60. Ernst, M.D.: Type annotations specification (jsr 308), `http://pag.csail.mit.edu/jsr308/` (September 12, 2008)
61. Boyland, J.: Why we should not add readonly to Java (yet). In: FTfJP, Glasgow, Scotland. Springer (July 2005)
62. Lucassen, J.M., Gifford, D.K.: Polymorphic effect systems. In: POPL, pp. 47–57 (January 1988)
63. Talpin, J.P., Jouvelot, P.: The type and effect discipline. In: LICS, pp. 162–173 (June 1992)

64. Nielson, F., Riis Nielson, H.: Type and Effect Systems. In: Olderog, E.-R., Steffen, B. (eds.) Correct System Design. LNCS, vol. 1710, pp. 114–136. Springer, Heidelberg (1999)
65. Müller, P., Poetzsch-Heffter, A.: Universes: A type system for alias and dependency control. Technical Report 279, Fernuniversität Hagen (2001)
66. Haack, C., Poll, E., Schäfer, J., Schubert, A.: Immutable Objects for a Java-Like Language. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 347–362. Springer, Heidelberg (2007)
67. Porat, S., Biberstein, M., Koved, L., Mendelson, B.: Automatic detection of immutable fields in Java. In: CASCON (November 2000)
68. Fähndrich, M., Leino, K.R.M.: Declaring and checking non-null types in an object-oriented language. In: OOPSLA, pp. 302–312. ACM Press, New York (2003)
69. Fähndrich, M., Xia, S.: Establishing object invariants with delayed types. In: Gabriel, R.P., Bacon, D.F., Lopes, C.V., Steele Jr., G.L. (eds.) OOPSLA, pp. 337–350. ACM Press (2007)
70. Qi, X., Myers, A.C.: Masked types for sound object initialization. In: Shao, Z., Pierce, B.C. (eds.) POPL, pp. 53–65. ACM Press (2009)
71. Summers, A.J., Müller, P.: Freedom before commitment - a lightweight type system for object initialisation. In: OOPSLA. ACM Press (2011)
72. Leino, K.R.M., Müller, P., Wallenburg, A.: Flexible Immutability with Frozen Objects. In: Shankar, N., Woodcock, J. (eds.) VSTTE 2008. LNCS, vol. 5295, pp. 192–208. Springer, Heidelberg (2008)
73. Dietl, W., Müller, P.: Universes: Lightweight Ownership for JML. Journal of Object Technology 4(8), 5–32 (2005)
74. Dietl, W., Drossopoulou, S., Müller, P.: Generic Universe Types. In: Ernst, E. (ed.) ECOOP 2007. LNCS, vol. 4609, pp. 28–53. Springer, Heidelberg (2007)
75. Müller, P., Rudich, A.: Ownership transfer in Universe Types. In: OOPSLA (2007)
76. Haack, C., Poll, E., Schäfer, J., Schubert, A.: Immutable Objects for a Java-Like Language. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 347–362. Springer, Heidelberg (2007)
77. Tschantz, M.S., Ernst, M.D.: Javari: Adding reference immutability to Java. In: OOPSLA (2005)
78. Zibin, Y., Potanin, A., Artzi, S., Kieżun, A., Ernst, M.D.: Object and reference immutability using Java generics. Technical Report MIT-CSAIL-TR-2007-018, MITCSAIL (2007)
79. Kniesel, G., Theisen, D.: JAC—access right based encapsulation for Java. Software — Practice and Experience (2001)
80. Skoglund, M., Wrigstad, T.: Alias control with read-only references. In: Sixth Conference on Computer Science and Informatics (March 2002)
81. Lu, Y., Potter, J.: Protecting representation with effect encapsulation. In: POPL (2006)
82. Noble, J., Vitek, J., Potter, J.: Flexible Alias Protection. In: Jul, E. (ed.) ECOOP 1998. LNCS, vol. 1445, pp. 158–185. Springer, Heidelberg (1998)
83. Igarashi, A., Viroli, M.: Variant parametric types: A flexible subtyping scheme for generics. ACM Transactions on Programming Languages and Systems (TOPLAS) 28(5), 795–847 (2006)
84. DeLine, R., Fähndrich, M.: Typestates for Objects. In: Odersky, M. (ed.) ECOOP 2004. LNCS, vol. 3086, pp. 465–490. Springer, Heidelberg (2004)