Javari: Adding Reference Immutability to Java

Matthew S. Tschantz Michael D. Ernst
MIT CSAIL
Cambridge, MA, USA
tschantz@mit.edu, mernst@csail.mit.edu

Abstract

This paper describes a type system that is capable of expressing and enforcing immutability constraints. The specific constraint expressed is that the abstract state of the object to which an immutable reference refers cannot be modified using that reference. The abstract state is (part of) the transitively reachable state: that is, the state of the object and all state reachable from it by following references. The type system permits explicitly excluding fields from the abstract state of an object. For a statically type-safe language, the type system guarantees reference immutability. If the language is extended with immutability downcasts, then run-time checks enforce the reference immutability constraints.

This research builds upon previous research in language support for reference immutability. Improvements that are new in this paper include distinguishing the notions of assignability and mutability; integration with Java 5's generic types and with multi-dimensional arrays; a mutability polymorphism approach to avoiding code duplication; type-safe support for reflection and serialization; and formal type rules and type soundness proof for a core calculus. Furthermore, it retains the valuable features of the previous dialect, including usability by humans (as evidenced by experience with 160,000 lines of Javari code) and interoperability with Java and existing JVMs.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features—*data types*; F.3.1 [**Logics and Meaning of Programs**]: Specifying and Verifying and Reasoning about Programs; D.1.5 [**Programming Techniques**]: Object-oriented Programming

General Terms

Languages, Theory, Experimentation

Keywords

`assignable`, immutability, Java, Javari, `mutable`, `readonly`, type system, verification

OOPSLA'05, October 16–20, 2005, San Diego, California, USA.

1. Introduction

The Javari programming language extends Java to allow programmers to specify and enforce reference immutability constraints. An immutable, or read-only, reference cannot be used to modify the object, including its transitive state, to which it refers. A type system enforcing reference immutability has a number of benefits: it can increase expressiveness; it can enhance program understanding and reasoning by providing explicit, machine-checked documentation; it can save time by preventing and detecting errors that would otherwise be very difficult to track down; and it can enable analyses and transformations that depend on compiler-verified properties.

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; reference immutability is more flexible, as it provides useful guarantees even about code that manipulates mutable objects. For example, many objects are modified during a construction phase but not thereafter. As another example, an interface can specify that a method that receives an immutable reference as a parameter does not modify the parameter through that reference, or that a caller does not modify a return value. Furthermore, a subsequent analysis can strengthen reference immutability into stronger guarantees, such as object immutability, where desired.

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 [5]. Despite this success, deficiencies of a previous proposal for the Javari language limit its applicability in practice. This paper provides a new design that retains the spirit of the previous proposal while revising the language to accommodate generic classes and arrays, to cleanly prevent code duplication, and to support reflection and serialization. Fur-

thermore, the new design clarifies the distinction between assignability and mutability, and we provide type rules and a soundness proof in the context of a core calculus based on Featherweight Generic Java, among other new contributions.

The rest of this document is organized as follows. Section 2 presents examples that motivate the need for reference immutability in a programming language. Section 3 describes the Javari language. Section 4 formalizes the type rules of Javari. Section 5 demonstrates how method templates can reduce code duplication. Section 6 discusses how the language handles reflection and serialization. Section 7 briefly discusses language features adopted from an earlier proposal, including full interoperability with Java, type-based analyses that build on reference immutability, handling of exceptions, and dynamic casts, an optional feature that substitutes run-time for compile-time checking at specific places in the program. Section 8 discusses related work, including a previous dialect of the Javari language. Section 9 concludes with a summary of contributions. Appendix A gives examples of each of Javari's assignability and mutability types being used. Finally, appendix B explains why Java's annotation mechanism is not expressive enough to encode Javari's constraints.

2. Motivation

Reference immutability provides a variety of benefits in different situations. Many other papers and books (see section 8) have justified the need for immutability constraints. The following simple examples suggest a few uses of immutability constraints and give a flavor of the Javari language.

Consider a voting system containing the following routine:

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

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.

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 approximately:

```
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.

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` [6]. This works, but is error-prone and 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.

A similar form of dangerous, mutation-permitting aliasing can occur when a data structure stores information passed to it (for instance, in a constructor) and a client retains a reference. Use of the `readonly` keyword again ensures that either the client's copy is read-only and cannot be modified, or else the data structure makes a copy, insulating it from changes performed by the client. In other words, the annotations force programmers to copy only when necessary.

As a final example, reference immutability can be used, in conjunction with a subsequent analysis, to establish the stronger guarantee of object immutability: a value is never modified, via any reference, if all references are immutable. For example, there is only one reference when an object is first constructed. As another simple example, some data structures must be treated as mutable when they are being initialized, but as immutable thereafter; an analysis can build upon Javari in order to make both the code and the reasoning simple.

```
Graph g1 = new Graph();
... construct cyclic graph g1 ...
// Suppose no aliases to g1 exist.
readonly Graph g = g1;
g1 = null;
```

3. Language design

In Java, each variable or expression has an *assignability* property, controlled by the `final` keyword, that determines whether it may be the lvalue (left-hand side) in an assignment. In Javari, each reference (non-primitive variable or expression) additionally has a *mutability* property that determines whether its abstract state may be changed (for example, by setting its fields). Both properties are specified in the source code, checked at compile time, and need

no run-time representation. The assignability and mutability properties determine whether various operations, such as reassignment and calling side-effecting methods, are permitted on a reference. Javari extends Java by providing additional constraints that may be placed on the assignability and mutability of a reference. Javari's keywords are those of Java, plus four more: `assignable` (the complement of `final`), `readonly` and its complement `mutable`, and `romaybe`, a syntactic convenience that reduces code duplication.

Sections 3.1 and 3.2 introduce the concepts of assignability and mutability, respectively, and discuss their application to local variables. Section 3.3 discusses the assignability and mutability of fields. Section 3.4 applies Javari's constructs to generic classes, and section 3.5 to arrays. Finally, section 3.6 summarizes Javari's syntax.

3.1 Assignability and final references

Assignability determines whether a reference may be reassigned. By default, a local variable is assignable: it may be reassigned. Java's `final` keyword makes a variable unassignable — it cannot be reassigned. Javari retains the `final` keyword, but provides greater control over the assignability of references via the `assignable` keyword (see section 3.3.2).

Assignability does not affect the type of the reference. Assignability constraints add no new types to the Java type hierarchy, and there is no type `final T` (for an existing type `T`).

```
final Date d = new Date(); // unassignable
      Date e = new Date(); // assignable
e = d; // OK
d = e; // error: d cannot be reassigned
```

Except in section 7.5, all errors noted in code examples are compile-time errors.

3.2 Mutability and read-only references

Mutation is any modification to an object's abstract state. The abstract state is (part of) the transitively reachable state, which is the state of the object and all state reachable from it by following references. It is important to provide transitive (deep) immutability guarantees in order to capture the full abstract state represented by a Java object. Clients of a method or other construct are typically interested in properties of the abstraction, not the concrete representation. (Javari provides ways to exclude selected fields from the abstract state; see section 3.3.2.)

Javari's `readonly` type modifier declares immutability constraints. A reference that is declared to be of `readonly` type cannot be used to mutate the object to which it refers. For example, suppose a variable `rodate` declared to have type `readonly Date`. A read-only reference can be used only to perform actions on the `Date` object that do not modify it:

```
readonly Date rodate; // readonly reference to a Date object
...
rodate.getMonth();    // OK
rodate.setYear(2005); // error
```

For every Java reference type `T`, `readonly T` is a valid Javari type and a supertype of `T`; see figure 1. A mutable reference may be used where a read-only reference is expected, because it has all the functionality of a read-only reference. A read-only reference may not be used where a mutable reference is expected, because it does not have all the functionality of a mutable reference: it cannot be used to modify the state of the object to which it refers.

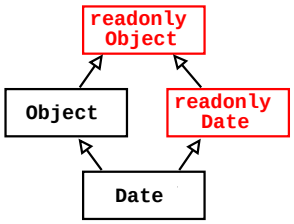

Figure 1: A portion of the Javari type hierarchy, which includes read-only and mutable versions of each Java reference type. Arrows connect subtypes to supertypes.

The type `readonly T` can be thought of as an interface that contains a subset of the methods of `T` (namely, those that do not mutate the object) and that is a supertype of `T`. However, Java interfaces cannot be used in place of the `readonly T` construct; see section 8.2 for details.

Given the type hierarchy shown in figure 1, Java's existing type-checking rules enforce that mutable methods cannot be called on read-only references and that the value referenced by a read-only variable cannot be copied to a non-read-only variable.

```
readonly    Date rodate = new Date(); // read-only Date
/*mutable*/ Date date   = new Date(); // mutable Date

rodate = date;                    // OK
rodate = (readonly Date) date;    // OK
date = rodate;                    // error
date = (Date) rodate;             // error: Java cast cannot make
                                  // a read-only reference mutable.
```

A read-only reference type can be used in a declaration of any variable, field, parameter, or method return type. Local variables, including method parameters and return types, are by default mutable (non-read-only). Primitive types, such as `int`, `float`, and `boolean`, are immutable — they contain no modifiable state. Thus, it is not meaningful to annotate them with `readonly`, and Javari's syntax prohibits it.

Note that `final` and `readonly` are orthogonal notions in a variable declaration: `final` makes the variable not assignable, but the object it references may still be mutable, while `readonly` makes the referenced object non-mutable (through that reference), but the variable may remain assignable. Using both keywords gives a variable whose transitively reachable state cannot be changed, except through a mutable aliasing reference.

3.2.1 *Read-only methods (`this` parameters)*

Just as `readonly` may be applied to any explicit formal parameter of a method, it may be applied to the implicit `this` parameter by writing `readonly` immediately following the parameter list. For example, an appropriate declaration for the `StringBuffer.charAt` method in Javari is:

```
public char charAt(int index) readonly { ... }
```

Such a method is called a read-only method. In the context of the method, `this` is `readonly`. Thus, it is a type error for a read-only method to change the state of the receiver, and it is a type error for a non-read-only method to be called through a read-only reference.

3.2.1.1 Overloading

Javari's method overloading mechanism is identical to that of Java, except that in Javari the mutability of the receiver is part of the signature, along with the method name and the parameter types.

Java performs two steps [19, §15.12] to determine which implementation of a method will be executed by a particular call site such as "`z = foo(x, y);`". The first step, overloading resolution, occurs at compile time, and the second step, dynamic dispatch, occurs at run time.

At compile time, Java determines the set of all matching signatures. A signature matches if its name is the same as that used at the call site, and each of its formal parameter types is a supertype of the declared types of the actual arguments at the call site. Java selects the most specific matching signature: the one whose formal parameters are equal to or subtypes of the corresponding formals of every other matching signature. If there are no matching signatures, or there is no single most specific matching signature, then Java issues a compile-time error.

At run time, Java uses dynamic dispatch to select an implementation of the signature that was identified at compile time. Dynamic dispatch selects the implementation whose receiver type is least, but is still at least as great as the run-time type of the receiver object.

Javari adopts the overloading resolution and dynamic dispatch mechanisms of Java without change.

```
void bar(readonly    Date) { ... }
void bar(/*mutable*/ Date) { ... } // overloaded
```

In Javari, the signature of a method includes not only the name of the method and the types of the parameters (as in Java), but also the mutability of the `this` parameter. Therefore, it is possible to have two methods declared in a single class with the same name and parameters, if one is read-only and the other is not. Such methods are resolved by overloading, not by overriding. Similarly, a read-only method declared in a subclass overloads (not overrides) a non-read-only method of same name and parameters declared in a superclass, and vice versa.

```
class Foo {
  void bar() readonly    { System.out.println(0); }

  // overloads, not overrides, bar() readonly:
  void bar() /*mutable*/ { System.out.println(1); }
}

/*mutable*/ Foo f  = new Foo();
readonly    Foo rf = f;

rf.bar();          // Prints 0
f.bar();           // Prints 1
```

Use of dynamic dispatch instead of overloading does not work. First, Javari does not require an implementation to have a run-time representation of the mutability of a reference — all checking can occur at compile time, unless a mutability downcast is used (see section 7.5). Second, for technical reasons, use of dynamic dispatch introduces either holes in the type system or unacceptably limits what actions a method can perform.

Overloading is also required to make accessors act like this-mutable fields, returning a read-only reference when invoked on a read-only receiver and a mutable reference when invoked on a mutable receiver. For instance, consider the `getValue` method of section 5.

Indiscriminate use of overloading can yield confusing code, both in Java and in Javari. If overloading based on immutability proves too troublesome in practice, we may restrict it in a future version of the language.

3.2.2 Immutable classes

A class or an interface can be declared to be immutable via the `readonly` modifier in its declaration.[1] This is a syntactic convenience: the class's non-static fields default to read-only and final, and its non-static methods (and, for inner classes, constructors) default to read-only. As a further syntactic convenience, every reference to an object of immutable type `T` is implicitly read-only. For example, `String` means the same thing as `readonly String` (the latter is forbidden, for syntactic simplicity), and that it is impossible to specify the type "mutable `String`".

```
readonly class String { ... }

/*readonly*/ String s1 = new String();
    readonly String s2 = new String();

s1 = s2;             // OK
s2 = s1;             // OK
```

Subclasses or subinterfaces of immutable classes and interfaces must be immutable. Any instance method inherited by such a class or interface must be read-only. Any instance field inherited by an immutable class or interface must be final and read-only or have been excluded from the class's abstract state (see section 3.3.2).

3.3 Fields: this-assignable and this-mutable

By default, a field of an object inherits its assignability and mutability from the reference through which the field is accessed. This default is called *this-assignability* and *this-mutability*. If the reference through which the field is accessed is read-only, then the field is unassignable (`final`) and read-only. If the reference through which the field is accessed is mutable, then the field is assignable and mutable. These defaults ensure that mutability is transitive by default. The defaults can be overridden through modifiers including `final`, `readonly`, `assignable` and `mutable` (The latter two are introduced in section 3.3.2.). The behavior of this-assignable and this-mutable fields is illustrated below.

```
class Cell {
  /*this-assignable this-mutable*/ Date d;
}

/*mutable*/ Cell  c; // mutable
readonly    Cell rc; // read-only

 c.d = new Date();   // OK: c.d is assignable
rc.d = new Date();   // error: rc.d is unassignable (final)

/*mutable*/ Date d1 =  c.d; // OK: c.d is mutable
/*mutable*/ Date d2 = rc.d; // error: rc.d is read-only

 c.d.setYear(2005);  // OK: c.d is mutable
rc.d.setYear(2005);  // error: rc.d is read-only
```

Only instance fields may be this-assignable or this-mutable. Other references (such as formal parameters and local variables) do not have a `this` to inherit their assignability from.

[1] A "read-only type" is `readonly T`, for some `T`. An "immutable type" is a primitive type, or a class or interface whose definition is marked with `readonly`.

3.3.1 Field accesses within methods

No special rules are needed for handling field accesses within method bodies. Within a read-only method, the declared type of `this` is read-only; therefore, all this-mutable fields are read-only and all this-assignable fields are unassignable. Within a non-read-only method, the declared type of `this` is mutable; therefore, all this-mutable fields are mutable and all this-assignable fields are assignable. These rules are demonstrated below.

```
class Cell {
  /*this-assignable this-mutable*/ Date d;

  /*mutable*/ Date foo() readonly {// this is readonly
    d = new Date();  // error: this.d is unassignable
    d.setYear(2005); // error: this.d is readonly
    return d;        // error: this.d is readonly
  }

  /*mutable*/ Date bar() /*mutable*/ {// this is mutable
    d = new Date();  // OK: this.d is assignable
    d.setYear(2005); // OK: this.d is mutable
    return d;        // OK: this.d is mutable
  }
}
```

3.3.2 Assignable and mutable fields

By default, fields are this-assignable and this-mutable. Under these defaults, all the fields are considered to be a part of the object's abstract state and, therefore, can not be modified through a read-only reference. The `assignable` and `mutable` keywords enable a programmer to exclude specific fields from the object's abstract state.

3.3.2.1 `assignable` fields

Declaring a field `assignable` specifies that the field may always be reassigned, even through a read-only reference. This can be useful for caching or specifying that the identity of a field is not a part of the object's abstract state. For example, `hashCode` is a read-only method, which does not modify the abstract state of the object. In order to record the hash code, a programmer can use the `assignable` keyword to exclude the field that the hash code is written to from the object's abstract state.

```
class Foo {
  assignable int hc;
  int hashCode() readonly {
    if (hc == 0) {
      hc = ... ;       // OK: hc is assignable
    }
    return hc;
  }
}
```

A this-mutable field accessed through a read-only reference is treated specially. As an rvalue (an expression in a value-expecting context [1]), it is read-only — it may be assigned to a read-only reference but not a mutable reference. However, as an lvalue (an expression in a location-expecting context, such as on the left side of an assignment [1]), it is mutable — it may be assigned with a mutable reference but not a read-only reference.

```
/** Assignable Cell. */
class ACell {
  assignable /*this-mutable*/ Date d;
}

readonly ACell rc = new ACell();
readonly Date  rd = new Date();
rc.d = rd;             // error: lvalue rc.d is mutable
rc.d.setYear(2005);    // error: rvalue rc.d is read-only
```

Without this asymmetry, there would be a loophole in the type system that could be used to convert a read-only reference to a mutable reference. This loophole is demonstrated below.

```
/** Assignable Cell. */
class ACell {
  assignable /*this-mutable*/ Date d;
}

/** Converts a read-only Date to a mutable date. */
static /*mutable*/ Date
convertReadonlyToMutable(readonly Date roDate) {
  /*mutable*/ ACell mutCell = new ACell();
  readonly ACell roCell = mutCell;
  roCell.d = roDate;                    // error
  /*mutable*/ Date mutDate = mutCell.d;
  return mutDate;
}
```

Mutable references, and this-mutable fields of the *same* object, may always be assigned to this-mutable fields.

3.3.2.2 `mutable` fields

The `mutable` keyword specifies that a field is mutable even when referenced through a read-only reference. A mutable field's value is not a part of the abstract state of the object (but the field's identity may be). For example, in the code below, `log` is declared `mutable` so that it may be mutated within read-only methods such as `hashCode`.

```
class Foo {
  final mutable List<String> log;

  int hashCode() readonly {
    log.add("entered hashCode()"); // OK: log is mutable
    ...
  }
}
```

3.4 Generic classes

In Java, the client of a generic class controls the type of any reference whose declared type is a type parameter. A client may instantiate a type parameter with any type that is equal to or a subtype of the declared bound. One can think of the type argument being directly substituted into the parameterized class wherever the corresponding type parameter appears.

Javari uses the same rules, extended in the natural way to account for the fact that Javari types include a mutability specification (figure 1). A use of a type parameter within the generic class body has the exact mutability with which the type parameter was instantiated. Generic classes require no special rules for the mutabilities of type arguments, and the defaults for local variables and fields are unchanged.

As with any local variable's type, type arguments to the type of a local variable may be mutable (by default) or read-only (through use of the `readonly` keyword). Below, four valid local variable, parameterized type declarations of `List` are shown. Note that the mutability of the parameterized type `List` does not affect the mutability of the type argument.

```
/*mutable*/ List</*mutable*/ Date> ld1; // add/rem./mut.
/*mutable*/ List<readonly    Date> ld2; // add/remove
readonly    List</*mutable*/ Date> ld3; // mutate
readonly    List<readonly    Date> ld4; // (neither)
```

As with any instance field's type, type arguments to the type of a field default to this-mutable, and this default can be overridden by declaring the type argument to be `readonly` or `mutable`:

```
class DateList {
  // 3 readonly lists whose elements have different mutability
  readonly List</*this-mutable*/ Date> lst;
  readonly List<readonly         Date> lst2;
  readonly List<mutable          Date> lst3;
}
```

There are no special rules for the handling of this-mutability when applied to a type argument. As in any other case, the mutability of a type with this-mutability is determined by the mutability of the object in which it appears (not the mutability of the parameterized class in which it might be a type argument).[2] In the case of `DateList` above, the mutability of `lst`'s elements is determined by the mutability of the reference to `DateList`, not by the mutability of `lst` itself. The following example illustrates this behavior.

```
/*mutable*/ Date d;
/*mutable*/ DateList mutDateList = new DateList();
readonly    DateList roDateList = mutDateList;

// The reference through which lst is accessed determines
// the mutability of the elements.
d = mutDateList.lst.get(0); // OK: elems are mutable
d = roDateList.lst.get(0);  // error: elems are readonly

readonly List</*mutable*/ Date> rld;
readonly List<readonly    Date> rlrd;

rld = mutdateList.lst;    // OK
rld = roDateList.lst;     // error: different type params
rlrd = mutDateList.lst;   // error: different type params
rlrd = roDateList.lst;    // OK
```

Within a parameterized class, a reference whose type is a type parameter cannot be declared `mutable`. If such a type parameter were instantiated with a read-only type, an illegal downcast would occur. Similarly, Javari provides no way to specify this-mutability for a reference whose type is a type parameter. However, a type parameter may be declared `readonly`, because such a declaration can only result in safe upcasts. These rules are demonstrated below.

```
class Foo<T extends readonly Object> {
  // a is not this-mutable. Its mutability is determined
  // solely by the type that T is instantiated with.
              T a; // OK
     readonly T b; // OK: can only result in upcasts
      mutable T c; // error: T can be instantiated with a
                   // read-only type that cannot be
                   // casted to a mutable type.
  this-mutable T d; // error: and not valid syntax
}
```

[2]There is no need for a modifier that specifies that the type parameter's mutability should be inherited from the mutability of the parameterized class. The declaration of a reference to a parameterized class specifies the parameterized class's mutability, and the declaration can specify the same mutability for the type argument. For example, a `List` where the mutability of the type parameter matches the mutability of the `List`'s `this` type can be declared as follows: `List<Date>` or `readonly List<readonly Date>`.

If a type parameter is declared to extend a mutable type, then augmenting a reference whose type is the type parameter to be this-mutable would be safe. No declaration about such a type parameter's mutability could result in an unsafe downcast from a read-only type to a mutable type. However, we do not believe that the benefits of such a declaration justify the syntactic complexity that would accompany it (such as an additional keyword).

```
// T cannot be instantiated with a read-only type because
// no read-only type extends mutable Object.
class Foo<T extends /*mutable*/ Object> {
              T a; // OK
     readonly T b; // OK
      mutable T c; // OK: not valid syntax, however,
                   // because T is already mutable
  this-mutable T d; // OK: not valid syntax, however
}
```

3.5 Arrays

As with generic container classes, a programmer may independently specify the mutability of each level of an array. As with any other local variable's type, each level of an array is mutable by default and may be declared read-only with the `readonly` keyword. As with any other field's type, each level may be declared mutable with the `mutable` keyword, read-only with the `readonly` keyword, or this-mutable by default. Parentheses may be used to specify to which level of an array a keyword is to be applied. Below, four valid array local variable declarations are shown.

```
                  Date [] ad1; // add/remove, mutate
        (readonly Date)[] ad2; // add/remove
readonly          Date [] ad3; // mutate
readonly (readonly Date)[] ad4; // no add/rem., no mutate
```

The above syntax can be applied to arrays of any dimensionality. For example, the type `(readonly Date[])[]` is a two-dimensional array with a read-only inner-array and that is otherwise mutable.

Java's arrays are covariant. To maintain type safety, the JVM performs a check when a object is stored to an array. To avoid a run-time representation of immutability, Javari does not allow covariance across the mutability of array element types.

```
          Date [] ad1;
(readonly Date)[] ad2;

ad2 = ad1; // error: arrays are not covariant over mutability
```

3.6 Summary

Javari enables a programmer to independently declare the assignability and mutability of a reference. Figure 2 summarizes the assignability and mutability keywords of Javari:

`final` declares a reference to be unassignable.

`assignable` declares a reference always to be assignable even if accessed through a read-only reference. Redundant for references other than instance fields.

`readonly` declares a reference to be read-only. Redundant for immutable types.

`mutable` declares a reference always to be mutable even if accessed through a read-only reference. Redundant for references other than instance fields. Cannot be applied to type parameters (see section 3.4).

Figure 3 briefly gives the semantics of the keywords. Appendix A gives examples of each of the 9 combinations of assignability and mutability.

Construct	Assignability			Mutability		
	assignable	unassignable	this-assignable	mutable	read-only	this-mutable
Instance fields	`assignable`	`final`	(default)	`mutable`	`readonly`	(default)
Static fields	(default)	`final`	N/A	(default)	`readonly`	N/A
Local variables	(default)	`final`	N/A	(default)	`readonly`	N/A
Formal parameters	(default)	`final`	N/A	(default)	`readonly`	N/A
Return values	N/A	N/A	N/A	(default)	`readonly`	N/A
`this`	N/A	(default)	N/A	(default)	`readonly`	N/A

Figure 2: Javari's keywords. "N/A" denotes assignabilities or mutabilities that are not valid for a given construct. "(default)" denotes that a given assignability or mutability is the default for that construct; no keyword is required, and redundant use of keywords is prohibited (a compile-time error), in order to reduce confusion. This-assignable and this-mutable can only be applied to instance fields because other references do not have a notion of `this` to inherit from. The mutability of `this` is declared after the parameter list.

Resolved assignability of `a.b`

Declared assignability of `b`	Resolved mutability of `a`	
	mutable	read-only
assignable	assignable	assignable
unassignable	unassignable	unassignable
this-assignable	assignable	unassignable

Resolved mutability of `a.b`

Declared mutability of `b`	Resolved mutability of `a`	
	mutable	read-only
mutable	mutable	mutable
read-only	read-only	read-only
this-mutable	assignable	read-only*

*mutable as an lvalue, read-only as an rvalue

Figure 3: Semantics of Javari's keywords: resolved type of the expression `a.b`, given the resolved type of `a` and the declared type of field `b`. Also see figure 9, which presents the same information in a different form.

```
T  ::=  X
     |  N
N  ::=  C<T̄>
L  ::=  class C<X̄ ◁ N̄> ◁ N {AF T̄ f̄; K M̄}
K  ::=  C(T̄ f̄) {super(f̄); this.f̄ = f̄;}
M  ::=  <X̄ ◁ N̄> T m(T̄ x̄){ return e; }
e  ::=  x
     |  e.f
     |  e.m<T̄>(ē)
     |  new N(ē)
     |  set e.f = e then e
AF ::=  final
     |  assignable
```

Figure 4: Lightweight Java (LJ) syntax. Changes from Featherweight Generic Java (FGJ) are indicated by boxes.

Javari is backward compatible with Java: any Java program that uses none of Javari's keywords is a valid Javari program, with the same semantics. Javari's defaults have been chosen to ensure this property.

4. Type rules

This section presents the key type rules for Javari in the context of a core calculus, Lightweight Javari, that captures the essential features of Javari. Lightweight Javari builds upon Featherweight Generic Java (FGJ) [20], a core calculus for Java including generic types. FGJ is a functional language: it has no notion of assignment. Therefore, Section 4.1 first introduces Lightweight Java (LJ), an extension of FGJ that builds on ClassicJava [15] to support field assignment and the `final` keyword. Then, Section 4.2 extends LJ to Lightweight Javari (LJR), which adds support for reference immutability.

4.1 Lightweight Java

Lightweight Java (LJ) extends FGJ to include field assignment, expressed via the "`set`" construct. Fields may be declared `final`; to make FGJ's syntax fully explicit, non-final fields must be declared `assignable`. LJ does not permit parameters to be reassigned: such an extension is straightforward and does not demonstrate any interesting aspects of the Javari type system. LJ omits casts because they are not important for demonstrating the assignability and mutability rules of Javari.

4.1.1 Syntax

The syntax of LJ is nearly identical to that of FGJ, with the exception of the new `set` expression and the assignability modifiers, `final` and `assignable`. The syntax of LJ is shown in figure 4.

The metavariable `C` ranges over (unparameterized) class names; `f` and `g` over field names; `AF` (assignability for fields) over assignability modifiers; `x` over variables; `e` over expressions; `K` over constructor declarations; and `M` over method declarations. `S`, `T`, `U`, and `V` range over types; `X`, `Y`, and `Z` over type variables; and `N`, `P`, and `Q` over nonvariable types.

$\overline{\texttt{x}}$ serves as shorthand for the (possibly empty) sequence $\texttt{x}_1 \ldots \texttt{x}_n$ with the appropriate syntax separating the elements. In cases such as $\overline{\texttt{C}}\ \overline{\texttt{f}}$, the items are grouped in pairs: $\texttt{C}_1\ \texttt{f}_1 \ldots \texttt{C}_n\ \texttt{f}_n$. Sequences are concatenated with a comma.

`this` is considered a special variable implicitly bound to the receiver of a method invocation.

The notation $\triangleleft$ stands for "extends" (LJ has no interfaces). A class is considered to have all the fields that are declared in its body and its superclasses' bodies. The field names of a class must be distinct from the field names of its superclasses — there is no field shadowing nor overloading.

Subtyping:

$$\Delta \vdash \mathtt{T} <: \mathtt{T} \text{ (S-Refl)}$$

$$\frac{\Delta \vdash \mathtt{S} <: \mathtt{T} \qquad \Delta \vdash \mathtt{T} <: \mathtt{U}}{\Delta \vdash \mathtt{S} <: \mathtt{U}} \text{ (S-Trans)}$$

$$\Delta \vdash \mathtt{X} <: \Delta(\mathtt{X}) \text{ (S-Var)}$$

$$\frac{\mathtt{class\ C<}\overline{\mathtt{X}} \lhd \overline{\mathtt{N}}\mathtt{>} \lhd \mathtt{N}\ \{\ldots\}}{\Delta \vdash \mathtt{C<}\overline{\mathtt{T}}\mathtt{>} <: [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{N}} \text{ (S-Class)}$$

Figure 5: Lightweight Java (LJ) subtyping. These rules are identical to those of Featherweight Generic Java (FGJ).

Every class is required to declare a single constructor. The form of the constructor must be:

$$\mathtt{C}(\overline{\mathtt{U}}\ \overline{\mathtt{g}}; \overline{\mathtt{T}}\ \overline{\mathtt{f}})\{\mathtt{super}(\overline{\mathtt{g}});\ \mathtt{this}.\overline{\mathtt{f}} = \overline{\mathtt{f}};\}$$

where $\overline{\mathtt{g}}$ are the fields of the superclass of C and $\overline{\mathtt{f}}$ are the fields declared in the body of C.

LJ introduces the set construct to FGJ. "set $\mathtt{e}_0.\mathtt{f} = \mathtt{e}_v$ then $\mathtt{e}_b$" reassigns the field f of the object to which $\mathtt{e}_0$ evaluates. The field's new value is the value to which $\mathtt{e}_v$ evaluates. The set expression then evaluates the body expression $\mathtt{e}_b$. The set expression's value is the value to which $\mathtt{e}_b$ evaluates. The set syntax was chosen to avoid the complications of allowing multiple expressions within a method. Method arguments and assignment expressions are evaluated left-to-right (see figure 7).

4.1.2 Static semantics

The subtyping rules of LJ are unchanged from FGJ and are shown in figure 5. The (reflexive, transitive) subtyping relationship is denoted by "$<:$". Δ is the type environment, a mapping from type variables to non-variable types.

The subtyping and reduction rules use the following auxiliary functions. The function *fields*(N) returns a sequence of triplets, $\overline{\mathtt{AF}}\ \overline{\mathtt{T}}\ \overline{\mathtt{f}}$, with the assignability modifier, type, and name of each of the fields of the nonvariable type N. The function *mtype*(m, N) returns the type of method m of the nonvariable type N. The type of a method is written as $\mathtt{<}\overline{\mathtt{X}} \lhd \overline{\mathtt{N}}\mathtt{>}\overline{\mathtt{T}} \to \mathtt{T}$ where $\overline{\mathtt{X}}$, with the bounds $\overline{\mathtt{N}}$, are the type parameters of the method, $\overline{\mathtt{T}}$ are the types of the method's parameters, and T is the return type of the method. Because there is no overloading in LJ, *mtype* does not need to know the types of the arguments to m. The function *mbody*(m<$\overline{\mathtt{V}}$>, N) returns the pair $\overline{\mathtt{x}}.\mathtt{e}$ where $\overline{\mathtt{x}}$ are the formal parameters to m in N and e is the body of the method. Finally, *override*(m, N, $\mathtt{<}\overline{\mathtt{Y}} \lhd \overline{\mathtt{P}}\mathtt{>}\overline{\mathtt{T}} \to \mathtt{T}$) declares that method m with type $\mathtt{<}\overline{\mathtt{Y}} \lhd \overline{\mathtt{P}}\mathtt{>}\overline{\mathtt{T}} \to \mathtt{T}$) correctly overrides any methods with the same name possessed by the nonvariable type N. Details and definitions of these auxiliary functions are provided in the FGJ paper.

With the exception of the new rule for the set construct, LJ's typing rules are little changed from those of FGJ. The typing rules are shown in figure 6. Γ is defined as the environment, a mapping from variables to types. $bound_\Delta(\mathtt{T})$ calculates the upper bound of T in type environment Δ: $\Delta(\mathtt{T})$ if T is a type parameter, or T if T is a nonvariable type. [a/b]c denotes the result of replacing b by a in c.

Expression typing:

$$\Delta;\Gamma \vdash \mathtt{x} : \Gamma(\mathtt{x}) \qquad \text{(T-Var)}$$

$$\frac{\Delta;\Gamma \vdash \mathtt{e}_0 : \mathtt{T}_0 \qquad \mathit{fields}(\mathit{bound}_\Delta(\mathtt{T}_0)) = \boxed{\overline{\mathtt{AF}}}\ \overline{\mathtt{T}}\ \overline{\mathtt{f}}}{\Delta;\Gamma \vdash \mathtt{e}_0.\mathtt{f}_i : \mathtt{T}_i} \text{ (T-Get)}$$

$$\frac{\begin{array}{c}\Delta;\Gamma \vdash \mathtt{e}_0 : \mathtt{T}_0 \\ \mathit{mtype}(\mathtt{m}, \mathit{bound}_\Delta(\mathtt{T}_0)) = \mathtt{<}\overline{\mathtt{Y}} \lhd \overline{\mathtt{P}}\mathtt{>}\overline{\mathtt{U}} \to \mathtt{U} \\ \Delta \vdash \overline{\mathtt{V}}\ \mathtt{ok} \qquad \Delta \vdash \overline{\mathtt{V}} <: [\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\overline{\mathtt{P}} \\ \Delta;\Gamma \vdash \overline{\mathtt{e}} : \overline{\mathtt{S}} \qquad \Delta \vdash \overline{\mathtt{S}} <: [\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\overline{\mathtt{U}}\end{array}}{\Delta;\Gamma \vdash \mathtt{e}_0.\mathtt{m<}\overline{\mathtt{V}}\mathtt{>}(\overline{\mathtt{e}}) : [\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\mathtt{U}} \text{ (T-Invk)}$$

$$\frac{\begin{array}{c}\Delta \vdash \mathtt{N}\ \mathtt{ok} \qquad \mathit{fields}(\mathtt{N}) = \boxed{\overline{\mathtt{AF}}}\ \overline{\mathtt{T}}\ \overline{\mathtt{f}} \\ \Delta;\Gamma \vdash \overline{\mathtt{e}} : \overline{\mathtt{S}} \qquad \Delta \vdash \overline{\mathtt{S}} <: \overline{\mathtt{T}}\end{array}}{\Delta;\Gamma \vdash \mathtt{new\ N}(\overline{\mathtt{e}}) : \mathtt{N}} \text{ (T-New)}$$

$$\frac{\begin{array}{c}\Delta;\Gamma \vdash \mathtt{e}_0 : \mathtt{T}_0 \\ \mathit{fields}(\mathit{bound}_\Delta(\mathtt{T}_0)) = \overline{\mathtt{AF}}\ \overline{\mathtt{T}}\ \overline{\mathtt{f}} \qquad \mathtt{AF}_i = \mathtt{assignable} \\ \Delta;\Gamma \vdash \mathtt{e}_v : \mathtt{T}_v \qquad \mathtt{T}_v <: \mathtt{T}_i \qquad \Delta;\Gamma \vdash \mathtt{e}_b : \mathtt{T}_b\end{array}}{\Delta;\Gamma \vdash \mathtt{set\ e}_0.\mathtt{f}_i = \mathtt{e}_v\ \mathtt{then\ e}_b : \mathtt{T}_b} \text{ (T-Set)}$$

Method typing:

$$\frac{\begin{array}{c}\Delta \vdash \overline{\mathtt{X}} <: \overline{\mathtt{N}},\ \overline{\mathtt{Y}} <: \overline{\mathtt{P}} \qquad \Delta \vdash \overline{\mathtt{T}},\ \mathtt{T},\ \overline{\mathtt{P}}\ \mathtt{ok} \\ \Delta;\overline{\mathtt{x}} : \overline{\mathtt{T}}, \mathtt{this} : \mathtt{C<}\overline{\mathtt{X}}\mathtt{>} \vdash \mathtt{e}_0 : \mathtt{S} \\ \Delta \vdash \mathtt{S} <: \mathtt{T} \qquad \mathtt{class\ C<}\overline{\mathtt{X}} \lhd \overline{\mathtt{N}}\mathtt{>} \lhd \mathtt{N}\ \{\ldots\} \\ \mathit{override}(\mathtt{m}, \mathtt{N}, \mathtt{<}\overline{\mathtt{Y}} \lhd \overline{\mathtt{P}}\mathtt{>}\overline{\mathtt{T}} \to \mathtt{T})\end{array}}{\mathtt{<}\overline{\mathtt{Y}} \lhd \overline{\mathtt{P}}\mathtt{>\ T\ m}(\overline{\mathtt{T}}\ \overline{\mathtt{x}})\{\ \mathtt{return\ e}_0;\ \}\ \mathtt{OK\ IN\ C<}\overline{\mathtt{X}} \lhd \overline{\mathtt{N}}\mathtt{>}} \text{ (T-Method)}$$

Class typing:

$$\frac{\begin{array}{c}\overline{\mathtt{X}} <: \overline{\mathtt{N}} \vdash \overline{\mathtt{N}},\ \mathtt{N},\ \overline{\mathtt{T}}\ \mathtt{ok} \\ \mathit{fields}(\mathtt{N}) = \overline{\mathtt{U}}\ \overline{\mathtt{g}} \qquad \overline{\mathtt{M}}\ \mathtt{OK\ IN\ C<}\overline{\mathtt{X}} \lhd \overline{\mathtt{N}}\mathtt{>} \\ \mathtt{K} = \mathtt{C}(\overline{\mathtt{U}}\ \overline{\mathtt{g}},\ \overline{\mathtt{T}}\ \overline{\mathtt{f}})\{\mathtt{super}(\overline{\mathtt{g}});\ \mathtt{this}.\overline{\mathtt{f}} = \overline{\mathtt{f}};\}\end{array}}{\mathtt{class\ C<}\overline{\mathtt{X}} \lhd \overline{\mathtt{N}}\mathtt{>} \lhd \mathtt{N}\ \{\boxed{\overline{\mathtt{AF}}}\ \overline{\mathtt{T}}\ \overline{\mathtt{f}};\ \mathtt{K}\ \overline{\mathtt{M}}\}\ \mathtt{OK}} \text{ (T-Class)}$$

Figure 6: Lightweight Java (LJ) typing rules. Changes from Featherweight Generic Java (FGJ) are indicated by boxes.

The judgment $\Delta \vdash \mathtt{T}\ \mathtt{ok}$ declares that type T is well formed under context Δ. A type is well formed if its type parameters respect the bounds placed on them in the class's declaration. The judgment M OK IN C declares that method declaration M is sound in the context of class C. The judgment C OK declares the class declaration of C to be sound.

4.1.3 Operational semantics

To support the assignment of fields, we introduce a store, $\mathcal{S}$, to the reduction rules. As in ClassicJava [15], the store is a mapping from an object to a pair containing the nonvariable type of the object and a field record. A field record, $\mathcal{F}$, is a mapping from field names to values.

The reduction rules are shown in figure 7. Each reduction rule is a relationship, $\langle \mathtt{e}, \mathcal{S}\rangle \longrightarrow \langle \mathtt{e}', \mathcal{S}'\rangle$, where e with store $\mathcal{S}$ reduces to $\mathtt{e}'$ with store $\mathcal{S}'$ in one step.

The addition of the assignment statement requires new reduction and congruence rules to be added to those of FGJ. The reduction rule for set binds the field to a new value, then evaluates the "then" part of the expression.

v ::= an object

Computation:

$$\frac{\mathcal{S}(v_1) = \langle \mathtt{N},\ \mathcal{F}\rangle \qquad \mathcal{F}(\mathtt{f}_i) = v_2}{\langle v_1.\mathtt{f}_i,\ \mathcal{S}\rangle \longrightarrow \langle v_2,\ \mathcal{S}\rangle}\ \text{(R-Field)}$$

$$\frac{\mathcal{S}(v) = \langle \mathtt{N},\ \mathcal{F}\rangle \qquad \mathtt{mbody(m<\overline{V}>,\ N)} = \overline{\mathtt{x}}.\mathtt{e}_0}{\langle v.\mathtt{m<\overline{V}>}(\overline{v}),\ \mathcal{S}\rangle \longrightarrow \langle [\overline{v}/\overline{\mathtt{x}},\ v/\mathtt{this}]\mathtt{e}_0,\ \mathcal{S}\rangle}\ \text{(R-Invk)}$$

$$\frac{v \notin \mathtt{dom}(\mathcal{S}) \qquad \mathcal{F} = [\overline{\mathtt{f}} \mapsto \overline{v}]}{\langle \mathtt{new\ N}(\overline{v}),\ \mathcal{S}\rangle \longrightarrow \langle v,\ \mathcal{S}[v \mapsto \langle \mathtt{N},\ \mathcal{F}\rangle]\rangle}\ \text{(R-New)}$$

$$\boxed{\frac{\mathcal{S}(v_1) = \langle \mathtt{N},\ \mathcal{F}\rangle}{\langle \mathtt{set}\ v_1.\mathtt{f}_i = v_2\ \mathtt{then}\ \mathtt{e}_b,\ \mathcal{S}\rangle \longrightarrow \langle \mathtt{e}_b,\ \mathcal{S}[v_1 \mapsto \langle \mathtt{N},\ \mathcal{F}[\mathtt{f}_i \mapsto v_2]\rangle]\rangle}}\ \text{(R-Set)}$$

Congruence:

$$\frac{\langle \mathtt{e}_0,\ \mathcal{S}\rangle \longrightarrow \langle \mathtt{e}_0',\ \mathcal{S}\rangle}{\langle \mathtt{e}_0.\mathtt{f},\ \mathcal{S}\rangle \longrightarrow \langle \mathtt{e}_0'.\mathtt{f},\ \mathcal{S}\rangle}\ \text{(RC-Field)}$$

$$\frac{\langle \mathtt{e}_0,\ \mathcal{S}\rangle \longrightarrow \langle \mathtt{e}_0',\ \mathcal{S}\rangle}{\langle \mathtt{e}_0.\mathtt{m<\overline{T}>}(\overline{\mathtt{e}}),\ \mathcal{S}\rangle \longrightarrow \langle \mathtt{e}_0'.\mathtt{m<\overline{T}>}(\overline{\mathtt{e}}),\ \mathcal{S}\rangle}\ \text{(RC-Invk-Recv)}$$

$$\frac{\langle \mathtt{e}_i,\ \mathcal{S}\rangle \longrightarrow \langle \mathtt{e}_i',\ \mathcal{S}\rangle}{\langle \boxed{v}.\mathtt{m<\overline{V}>}(\boxed{\overline{v}}, \mathtt{e}_i, \overline{\mathtt{e}}),\ \mathcal{S}\rangle \longrightarrow \langle \boxed{v}.\mathtt{m<\overline{V}>}(\boxed{\overline{v}}, \mathtt{e}_i', \overline{\mathtt{e}}),\ \mathcal{S}\rangle}\ \text{(RC-Invk-Arg)}$$

$$\frac{\langle \mathtt{e}_i,\ \mathcal{S}\rangle \longrightarrow \langle \mathtt{e}_i',\ \mathcal{S}\rangle}{\langle \mathtt{new\ N}(\boxed{\overline{v}}, \mathtt{e}_i, \overline{\mathtt{e}}),\ \mathcal{S}\rangle \longrightarrow \langle \mathtt{new\ N}(\boxed{\overline{v}}, \mathtt{e}_i', \overline{\mathtt{e}}),\ \mathcal{S}\rangle}\ \text{(RC-New-Arg)}$$

$$\boxed{\frac{\langle \mathtt{e}_0,\ \mathcal{S}\rangle \longrightarrow \langle \mathtt{e}_0',\ \mathcal{S}\rangle}{\langle \mathtt{set}\ \mathtt{e}_0.f = \mathtt{e}_v\ \mathtt{then}\ \mathtt{e}_b,\ \mathcal{S}\rangle \longrightarrow \langle \mathtt{set}\ \mathtt{e}_0'.f = \mathtt{e}_v\ \mathtt{then}\ \mathtt{e}_b,\ \mathcal{S}\rangle}}\ \text{(RC-Set-LHS)}$$

$$\boxed{\frac{\langle \mathtt{e}_v,\ \mathcal{S}\rangle \longrightarrow \langle \mathtt{e}_v',\ \mathcal{S}\rangle}{\langle \mathtt{set}\ v.f = \mathtt{e}_v\ \mathtt{then}\ \mathtt{e}_b,\ \mathcal{S}\rangle \longrightarrow \langle \mathtt{set}\ v.f = \mathtt{e}_v'\ \mathtt{then}\ \mathtt{e}_b,\ \mathcal{S}\rangle}}\ \text{(RC-Set-RHS)}$$

Figure 7: Lightweight Java (LJ) reduction rules. Unlike Featherweight Generic Java (FGJ), LJ's reduction rules contain a store. Other changes from FGJ are indicated by boxes.

4.1.4 Properties

LJ can be shown to be type sound and to obey the assignment rules of `final`. If a term is well typed and reduces to a normal form, an expression that cannot reduce any further and, therefore, is an object, v, then it is a value of a subtype of the original term's type. Put differently, an evaluation cannot go wrong, which in our model means getting stuck.

Theorem 1 (LJ Type Soundness). *If* $\emptyset;\emptyset \vdash \mathtt{e} : \mathtt{T}$ *and* $\langle \mathtt{e},\ \mathcal{S}\rangle \rightarrow^* \langle \mathtt{e}',\ \mathcal{S}'\rangle$ *with* $\mathtt{e}'$ *a normal form, then* $\mathtt{e}'$ *is a value* v *such that* $\mathcal{S}'(v) = \langle \mathtt{N},\ \mathcal{F}\rangle$ *and* $\emptyset \vdash \mathtt{N} <: \mathtt{T}$.

The soundness of LJ can be proved using the standard technique of subject reduction and progress theorems. The reduction theorem states that each step taken in the evaluation preserves the type correctness of the expression-store pair. During each step of the reduction, the environment and type environment must be consistent, $\vdash_\sigma$, with the store:

$$\begin{aligned}
&\Delta;\Gamma \vdash_\sigma \mathcal{S} \iff \\
&\quad \mathcal{S}(v) = \langle \mathtt{N},\ \mathcal{F}\rangle \Rightarrow \\
\Sigma_1 :&\quad \Delta \vdash \mathtt{N} <: \Gamma(v) \\
\Sigma_2 :&\quad \text{and } \mathrm{dom}(\mathcal{F}) = \{\mathtt{f} \mid \mathtt{AF\ T\ f} \in \mathit{fields}(\mathtt{N})\} \\
\Sigma_3 :&\quad \text{and } \mathrm{rng}(\mathcal{F}) \subseteq \mathrm{dom}(\mathcal{S}) \\
\Sigma_4 :&\quad \text{and } (\mathcal{F}(\mathtt{f}) = v' \text{and } \mathtt{AF\ T\ f} \in \mathit{fields}(\mathtt{N}) \\
&\qquad \Rightarrow ((\mathcal{S}(v') = \langle \mathtt{N}',\ \mathcal{F}'\rangle) \Rightarrow \Delta \vdash \mathtt{N}' <: \mathtt{T})) \\
\Sigma_5 :&\quad \text{and } v \in \mathrm{dom}(\Gamma) \Rightarrow v \in \mathrm{dom}(\mathcal{S}) \\
\Sigma_6 :&\quad \text{and } \mathrm{dom}(\mathcal{S}) \subseteq \mathrm{dom}(\Gamma)
\end{aligned}$$

Theorem 2 (LJ Subject Reduction). *If* $\Delta;\Gamma \vdash e : \mathtt{T}$, $\Delta;\Gamma \vdash_\sigma \mathcal{S}$, *and* $\langle \mathtt{e},\ \mathcal{S}\rangle \longrightarrow \langle \mathtt{e}',\ \mathcal{S}'\rangle$, *then there exists a* Γ' *such that* $\Delta;\Gamma' \vdash \mathtt{e}' : \mathtt{T}$ *and* $\Delta';\Gamma' \vdash_\sigma \mathcal{S}'$.

Proof [38]. By induction on the derivation of $\langle \mathtt{e},\ \mathcal{S}\rangle \longrightarrow \langle \mathtt{e}',\ \mathcal{S}'\rangle$ with a case analysis on the reduction rule used. For each case, we construct the new environment Γ' and show that (1) $\Delta;\Gamma' \vdash \mathtt{e}' : \mathtt{T}$ and (2) $\Delta';\Gamma' \vdash_\sigma \mathcal{S}$. □

Theorem 3 (LJ Progress). *If* $\Delta;\Gamma \vdash \mathtt{e} : \mathtt{T}$ *and* $\Delta;\Gamma \vdash_\sigma \mathcal{S}$, *then either* `e` *is a value or there exists an* $\langle \mathtt{e}',\ \mathcal{S}'\rangle$ *such that* $\langle \mathtt{e},\ \mathcal{S}\rangle \longrightarrow \langle \mathtt{e}',\ \mathcal{S}'\rangle$.

Proof [38]. The proof is by analysis of the possible cases for the current redex in `e` (in the case that `e` is not a value). □

The Type Soundness theorem is immediate from the Subject Reduction and Progress theorems.

Additionally, LJ can be shown to obey the rules of `final` references.

Theorem 4 (LJ Assignment Soundness). *Outside of an object's constructor, assignments may only be made to* `assignable` *fields.*

Proof. Immediate from T-Set. □

4.2 Lightweight Javari

We add Javari's concept of reference immutability to LJ to create the language Lightweight Javari (LJR). In LJR, every type is modified by one of the mutability modifiers: `readonly`, `mutable`, or (for field types only) `this-mutable`. In addition to `final` and `assignable`, LJR also allows fields be marked as this-assignable with the `this-assignable` keyword.

4.2.1 Syntax

The syntax of LJR is shown in figure 8. `C` continues to range over class names. The other type meta-variables range over both read-only and mutable types. Thus, a type is a pair, `ML B`, consisting of a mutability and a "base" type. `B` ranges over base types (which may be a parameterized class but has no mutability information) and `ML` (mutability for locals) ranges over `readonly` and `mutable`. `F` and `G` range over

```
T  ::=  X
    |   N
N  ::=  [ML] B
B  ::=  C<T̄>
[F] ::=  MF B
    |   X
L  ::=  class C<X̄ ◁ N̄> ◁ N {AF [F] f̄; K M̄}
K  ::=  C(T̄ f̄) {super(f̄); this.f = f̄;}
M  ::=  <X̄ ◁ N̄> T m(T̄ x̄) [ML] { return e; }
e  ::=  x
    |   e.f
    |   e.m<T̄>(ē)
    |   new B(ē)
    |   set e.f = e then e
AF ::=  final
    |   assignable
    |   [this-assignable]
[MF] ::=  ML
    |   this-mutable
[ML] ::=  mutable
    |   readonly
```

Figure 8: Lightweight Javari (LJR) syntax. Changes from Lightweight Java (LJ, figure 4) are indicated by boxes.

field type declarations, including those with this-mutability. `MF` (mutability for fields) ranges over the mutabilities that may be applied to a nonvariable type that is the type of a field. A mutability modifier may not be applied to a field whose type is a type parameter (see section 3.4). Thus, `F` takes the form of either a type parameter, `X`, or a nonvariable type with a field mutability modifier, `MF B`. All type metavariables other than `F` and `G` may not have this-mutability. `AF` ranges over the assignabilities that a field may be declared to have: `assignable`, `final`, and `this-assignable`.

4.2.2 Static semantics

Fields that are declared to be this-mutable are resolved to have either a read-only or mutable type based on the mutability of the reference through which the field is accessed. Also, this-assignable fields are resolved to be either final or assignable using the mutability of the reference through which the field is accessed. A type that is declared `readonly` or `mutable` trivially resolves to the read-only and mutable types, respectively. Similarly, fields that are declared `final` or `assignable` are trivially resolved.

The functions *assignability*(`AF`, `ML`) and *mutability*(`MF`, `ML`) (shown in figure 9) are used to resolve the assignability and mutability of a field. The first parameter to either function is the keyword with which the field is declared. The second parameter is the mutability of the reference through which the field is accessed or the declared bound of a reference whose declared type is a type parameter.

Additional subtyping rules must be added to those of LJ to handle the subtyping relationship between mutable and read-only types. The new rules are shown in figure 10. The subtyping rules of LJ are changed to be in terms of base types, `B`. This-mutability is not a part of the type hierarchy

Assignability resolving:

$$\mathit{assignability}(\texttt{assignable},\ \texttt{ML}) = \texttt{assignable}$$
$$\mathit{assignability}(\texttt{final},\ \texttt{ML}) = \texttt{final}$$
$$\mathit{assignability}(\texttt{this-assignable},\ \texttt{mutable}) = \texttt{assignable}$$
$$\mathit{assignability}(\texttt{this-assignable},\ \texttt{readonly}) = \texttt{final}$$

Mutability resolving:

$$\mathit{mutability}(\texttt{mutable},\ \texttt{ML}) = \texttt{mutable}$$
$$\mathit{mutability}(\texttt{readonly},\ \texttt{ML}) = \texttt{readonly}$$
$$\mathit{mutability}(\texttt{this-mutable},\ \texttt{mutable}) = \texttt{mutable}$$
$$\mathit{mutability}(\texttt{this-mutable},\ \texttt{readonly}) = \texttt{readonly}$$

Figure 9: Lightweight Javari (LJR): Resolving assignability and mutability. Also see figure 3, which presents the same information in a different form.

Subtyping:

$$\frac{\Delta \ \vdash\ \texttt{B}_1 <: \texttt{B}_2}{\Delta \ \vdash\ \texttt{ML B}_1 \ <: \texttt{ML B}_2}$$

$$\Delta \ \vdash\ \texttt{mutable B} <: \texttt{readonly B}$$

Figure 10: Lightweight Javari (LJR) additional subtyping rules.

and, thus, is not shown in the subtyping rules. The mutability of a field declared `this-mutable` must be resolved before subtyping relations may be calculated.

The type rules of LJR are shown in figure 11. The rules of LJ are augmented to check that the mutability of an expression is correct for its context. For example, the mutability of the receiver is a part of a method's signature; thus, a non-read-only method cannot be invoked on a read-only receiver.

Similar to $\mathit{bound}_\Delta(\texttt{T})$ we define $\mathit{fbound}_\Delta(\texttt{F})$ to calculate the upper bound of `F` in type environment Δ: $\Delta(\texttt{F})$ if `F` is a type parameter or `MF B` if `F` is a nonvariable type.

As an lvalue, a field whose type is this-mutable must be treated as mutable even if it was accessed through a read-only reference (see section 3.3.2). In the T-Set rule for field assignments, the mutability of the lvalue reference (through which the field being set is accessed) is always treated as `mutable`. Additionally, the T-Set-This assignment rule allows two fields of the same object whose types are this-mutable to be assigned to one another.

The rules for `new` expressions and constructor typing require that the object created is mutable because the object may be assigned to a mutable reference. Therefore, when calculating the mutability of this-mutable fields, the mutability of the reference through which the fields are accessed is given as `mutable`.

Auxiliary functions that are changed from LJ's are shown in figure 12. The functions for method type lookup and valid method overriding must be modified because the mutability of the receiver is part of a method's signature. In the case of method lookup, the function now returns the mutability of the receiver as a part of the method's type. The rule for

Expression typing:

$$\Delta;\Gamma \vdash \mathtt{x} : \Gamma(\mathtt{x}) \qquad \text{(RT-VAR)}$$

$$\frac{\begin{array}{c}\Delta;\Gamma \vdash \mathtt{e}_0 : \mathtt{T}_0 \\ bound_\Delta(\mathtt{T}_0) = \boxed{\mathtt{ML}_0}\ \mathtt{B}_0 \qquad fields(\mathtt{B}_0) = \overline{\mathtt{AF}}\ \boxed{\overline{\mathtt{F}}}\ \overline{\mathtt{f}} \\ fbound_\Delta(\mathtt{F}_i) = \boxed{\mathtt{MF}_i}\ \mathtt{B}_i \qquad \boxed{\mathtt{ML}_i = mutability(\mathtt{MF}_i,\ \mathtt{ML}_0)}\end{array}}{\Delta;\Gamma \vdash \mathtt{e}_0.\mathtt{f}_i : \boxed{\mathtt{ML}_i}\ \mathtt{B}_i} \qquad \text{(RT-GET)}$$

$$\frac{\begin{array}{c}\Delta;\Gamma \vdash \mathtt{e}_0 : \mathtt{T}_0 \qquad bound_\Delta(\mathtt{T}_0) = \boxed{\mathtt{ML}_0}\ \mathtt{B}_0 \\ mtype(\mathtt{m}, \mathtt{B}_0) = \ \mathtt{<}\overline{\mathtt{Y}} \lhd \overline{\mathtt{P}}\mathtt{>}\overline{\mathtt{U}}\ \boxed{\mathtt{ML}_m} \to \mathtt{U} \\ \boxed{\mathtt{ML}_0\ \mathtt{B}_0 <: \mathtt{ML}_m\ \mathtt{B}_0} \qquad \Delta \vdash \overline{\mathtt{V}}\ \mathtt{ok} \\ \Delta \vdash \overline{\mathtt{V}} <: [\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\overline{\mathtt{P}} \qquad \Delta;\Gamma \vdash \overline{\mathtt{e}} : \overline{\mathtt{S}} \qquad \Delta \vdash \overline{\mathtt{S}} <: [\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\overline{\mathtt{U}}\end{array}}{\Delta;\Gamma \vdash \mathtt{e}_0.\mathtt{m}\mathtt{<}\overline{\mathtt{V}}\mathtt{>}(\overline{\mathtt{e}}) : [\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\mathtt{U}} \qquad \text{(RT-INVK)}$$

$$\frac{\begin{array}{c}\Delta \vdash \mathtt{B}\ \mathtt{ok} \qquad fields(\mathtt{B}) = \overline{\mathtt{AF}}\ \boxed{\overline{\mathtt{F}}}\ \overline{\mathtt{f}} \\ fbound_\Delta(\overline{\mathtt{F}}) = \boxed{\overline{\mathtt{MF}}}\ \overline{\mathtt{B}} \qquad \boxed{\overline{\mathtt{ML}} = mutability(\overline{\mathtt{MF}},\ \mathtt{mutable})} \\ \Delta;\Gamma \vdash \overline{\mathtt{e}} : \overline{\mathtt{S}} \qquad \Delta \vdash \overline{\mathtt{S}} <: \boxed{\overline{\mathtt{ML}}}\ \overline{\mathtt{B}}\end{array}}{\Delta;\Gamma \vdash \mathtt{new}\ \mathtt{B}(\overline{\mathtt{e}}) : \mathtt{mutable}\ \mathtt{B}} \qquad \text{(RT-NEW)}$$

$$\frac{\begin{array}{c}\Delta;\Gamma \vdash \mathtt{e}_0 : \mathtt{T}_0 \\ bound_\Delta(\mathtt{T}_0) = \boxed{\mathtt{ML}_0}\ \mathtt{B}_0 \qquad fields(\mathtt{B}_0) = \boxed{\overline{\mathtt{AF}}}\ \boxed{\overline{\mathtt{F}}}\ \overline{\mathtt{f}} \\ \boxed{assignability(\mathtt{AF}_i,\ \mathtt{ML}_0) = \mathtt{assignable}} \\ fbound_\Delta(\mathtt{F}_i) = \boxed{\mathtt{MF}_i}\ \mathtt{B}_i \\ \boxed{\mathtt{ML}_i = mutability(\mathtt{MF}_i,\ \mathtt{mutable})} \\ \Delta;\Gamma \vdash \mathtt{e}_v : \mathtt{T}_v \qquad \Delta \ \vdash\ \mathtt{T}_v <: \mathtt{ML}_i\ \mathtt{B}_i \qquad \Delta;\Gamma \vdash \mathtt{e}_b : \mathtt{T}_b\end{array}}{\Delta;\Gamma \vdash \mathtt{set}\ \mathtt{e}_0.\mathtt{f}_i = \mathtt{e}_v\ \mathtt{then}\ \mathtt{e}_b : \mathtt{T}_b} \qquad \text{(RT-SET)}$$

$$\frac{\begin{array}{c}\Delta;\Gamma \vdash \mathtt{this} : \mathtt{ML}_0\ \mathtt{C}\mathtt{<}\overline{\mathtt{X}}\mathtt{>} \qquad fields(\mathtt{C}\mathtt{<}\overline{\mathtt{X}}\mathtt{>}) = \overline{\mathtt{AF}}\ \boxed{\overline{\mathtt{F}}}\ \overline{\mathtt{f}} \\ \boxed{assignability(\mathtt{AF}_i,\ \mathtt{ML}_0) = \mathtt{assignable}} \\ fbound_\Delta(\overline{\mathtt{F}}) = \boxed{\overline{\mathtt{MF}}}\ \overline{\mathtt{B}} \qquad \boxed{\mathtt{MF}_i = \mathtt{this\text{-}mutable}} \\ \boxed{\mathtt{MF}_j = \mathtt{this\text{-}mutable}} \qquad \Delta\ \vdash\ \mathtt{B}_j <: \mathtt{B}_i \qquad \Delta;\Gamma \vdash \mathtt{e}_b : \mathtt{T}_b\end{array}}{\Delta;\Gamma \vdash \mathtt{set}\ \mathtt{this}.\mathtt{f}_i = \mathtt{this}.\mathtt{f}_j\ \mathtt{then}\ \mathtt{e}_b : \mathtt{T}_b} \qquad \text{(RT-SET-THIS)}$$

Method typing:

$$\frac{\begin{array}{c}\Delta \vdash \overline{\mathtt{X}} <: \overline{\mathtt{N}},\ \overline{\mathtt{Y}} <: \overline{\mathtt{P}} \\ \Delta \vdash \overline{\mathtt{T}},\ \mathtt{T},\ \overline{\mathtt{P}}\ \mathtt{ok} \qquad \Delta;\overline{\mathtt{x}} : \overline{\mathtt{T}}, \mathtt{this} : \boxed{\mathtt{ML}}\ \mathtt{C}\mathtt{<}\overline{\mathtt{X}}\mathtt{>}\ \vdash \mathtt{e}_0 : \mathtt{S} \\ \Delta \vdash \mathtt{S} <: \mathtt{T} \qquad \mathtt{class}\ \mathtt{C}\mathtt{<}\overline{\mathtt{X}} \lhd \overline{\mathtt{N}}\mathtt{>} \lhd \mathtt{B}\ \{\ldots\} \\ override(\mathtt{m}, \mathtt{B}, \mathtt{<}\overline{\mathtt{Y}} \lhd \overline{\mathtt{P}}\mathtt{>}\overline{\mathtt{T}}\ \boxed{\mathtt{ML}} \to \mathtt{T})\end{array}}{\mathtt{<}\overline{\mathtt{Y}} \lhd \overline{\mathtt{P}}\mathtt{>}\ \mathtt{T}\ \mathtt{m}(\overline{\mathtt{T}}\ \overline{\mathtt{x}})\ \boxed{\mathtt{ML}}\ \{\mathtt{return}\ \mathtt{e}_0;\ \}\ \mathtt{OK\ IN\ C}\mathtt{<}\overline{\mathtt{X}} \lhd \overline{\mathtt{N}}\mathtt{>}} \qquad \text{(RT-METHOD)}$$

Class typing:

$$\frac{\begin{array}{c}fbound_\Delta(\overline{\mathtt{F}_c}) = \boxed{\overline{\mathtt{MF}_c}}\ \overline{\mathtt{B}_c} \\ \boxed{\overline{\mathtt{ML}_c} = mutability(\overline{\mathtt{MF}_c},\ \mathtt{mutable})} \qquad fields(\mathtt{B}) = \overline{\mathtt{AF}_b}\ \overline{\mathtt{G}_b}\ \overline{\mathtt{g}_b} \\ fbound_\Delta(\overline{\mathtt{G}_b}) = \overline{\mathtt{MF}_b}\ \overline{\mathtt{B}_b} \qquad \boxed{\overline{\mathtt{ML}_b} = mutability(\overline{\mathtt{MF}_b},\ \mathtt{mutable})} \\ \mathtt{K} = \mathtt{C}(\boxed{\overline{\mathtt{ML}_b}}\ \overline{\mathtt{B}_b}\ \overline{\mathtt{g}},\ \boxed{\overline{\mathtt{ML}_c}}\ \overline{\mathtt{B}_c}\ \overline{\mathtt{f}})\{\mathtt{super}(\overline{\mathtt{g}});\ \mathtt{this}.\overline{\mathtt{f}} = \overline{\mathtt{f}};\} \\ \overline{\mathtt{X}} <: \overline{\mathtt{N}} \vdash \overline{\mathtt{N}},\ \mathtt{B},\ \overline{\mathtt{B}_c}\ \mathtt{ok} \qquad \overline{\mathtt{M}}\ \mathtt{OK\ IN\ C}\mathtt{<}\overline{\mathtt{X}} \lhd \overline{\mathtt{N}}\mathtt{>}\end{array}}{\mathtt{class}\ \mathtt{C}\mathtt{<}\overline{\mathtt{X}} \lhd \overline{\mathtt{N}}\mathtt{>} \lhd \mathtt{B}\ \{\overline{\mathtt{AF}_c}\ \overline{\mathtt{F}_c}\ \overline{\mathtt{f}};\ \mathtt{K}\ \overline{\mathtt{M}}\}\ \mathtt{OK}} \qquad \text{(RT-CLASS)}$$

Figure 11: Lightweight Javari (LJR) typing rules. Changes from Lightweight Java (LJ, figure 6) are indicated by boxes.

Method type lookup:

$$\frac{\mathtt{class}\ \mathtt{C}\mathtt{<}\overline{\mathtt{X}} \lhd \overline{\mathtt{N}}\mathtt{>} \lhd \mathtt{B}\{\overline{\mathtt{AF}}\ \boxed{\overline{\mathtt{F}}}\ \overline{\mathtt{f}};\ \mathtt{K}\ \overline{\mathtt{M}}\} \qquad \mathtt{<}\overline{\mathtt{Y}} \lhd \overline{\mathtt{P}}\mathtt{>}\ \mathtt{U}\ \mathtt{m}(\overline{\mathtt{U}}\ \overline{\mathtt{x}})\ \boxed{\mathtt{ML}}\ \{\ \mathtt{return}\ \mathtt{e};\ \} \in \overline{\mathtt{M}}}{mtype(\mathtt{m}, \mathtt{C}\mathtt{<}\overline{\mathtt{T}}\mathtt{>}) = [\overline{\mathtt{T}}/\overline{\mathtt{X}}](\mathtt{<}\overline{\mathtt{Y}} \lhd \overline{\mathtt{P}}\mathtt{>}\overline{\mathtt{U}}\ \boxed{\mathtt{ML}} \to \mathtt{U})}$$

$$\frac{\mathtt{class}\ \mathtt{C}\mathtt{<}\overline{\mathtt{X}} \lhd \overline{\mathtt{N}}\mathtt{>} \lhd \mathtt{B}\{\overline{\mathtt{AF}}\ \boxed{\overline{\mathtt{F}}}\ \overline{\mathtt{f}};\ \mathtt{K}\ \overline{\mathtt{M}}\} \qquad \mathtt{m} \notin \overline{\mathtt{M}}}{mtype(\mathtt{m}, \mathtt{C}\mathtt{<}\overline{\mathtt{T}}\mathtt{>}) = mtype(\mathtt{m}, [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{B})}$$

Valid method overriding:

$$\frac{\begin{array}{c}mtype(\mathtt{m},\ \mathtt{B}) = \mathtt{<}\overline{\mathtt{Z}} \lhd \overline{\mathtt{Q}}\mathtt{>}\overline{\mathtt{U}}\ \boxed{\mathtt{ML}_1} \to \mathtt{U}_0 \text{ implies} \\ \overline{\mathtt{P}}, \overline{\mathtt{T}} = [\overline{\mathtt{Y}}/\overline{\mathtt{Z}}](\overline{\mathtt{Q}}, \overline{\mathtt{U}}) \text{ and } \overline{\mathtt{Y}} <: \overline{\mathtt{P}} \vdash \mathtt{T}_0 <: [\overline{\mathtt{Y}}/\overline{\mathtt{Z}}]\mathtt{U}_0 \text{ and } \boxed{\mathtt{ML}_1 = \mathtt{ML}_0}\end{array}}{override(\mathtt{m},\ \mathtt{B},\ \mathtt{<}\overline{\mathtt{Y}} \lhd \overline{\mathtt{P}}\mathtt{>}\overline{\mathtt{T}}\ \boxed{\mathtt{ML}_0} \to \mathtt{T}_0)}$$

Figure 12: Lightweight Javari (LJR) auxiliary functions. Changes from Lightweight Java (LJ) are indicated by boxes.

valid method overriding now checks that the mutability of the receiver in the overriding method matches the mutability of the receiver in the overridden method. Because LJ and LJR do not allow method overloading, there cannot be a read-only and a non-read-only version of a method.

4.2.3 Operational semantics

The reduction rules for LJR are unchanged from those of LJ (see figure 7) with the exception that the store, `new` expression, and *mbody* function operate on base types, `B`, which do not include mutability information.

4.2.4 Properties

LJR has a similar Type Soundness theorem as LJ. It is modified slightly due to the store operating on base types without mutability information.

THEOREM 5 (LJR TYPE SOUNDNESS). *If* $\emptyset;\emptyset \vdash \mathtt{e} : \mathtt{ML\ B}$ *and* $\langle \mathtt{e},\ \mathcal{S}\rangle \to^* \langle \mathtt{e}',\ \mathcal{S}'\rangle$ *with* $\mathtt{e}'$ *a normal form, then* $\mathtt{e}'$ *is a value* v *such that* $\mathcal{S}'(v) = \langle \mathtt{B}',\ \mathcal{F}\rangle$ *and* $\emptyset \vdash \mathtt{B}' <: \mathtt{B}$.

Note that the type soundness theorem is in terms of base types since there is no run-time concept of mutability. LJR's type soundness theorem is proved using the same reduction and progress theorems as LJ. The proofs of these theorems [38] are similar to those of LJ.

LJR's mutability constraints also allow a theorem about which state of an object, when referenced through a read-only reference, is protected from modification. This state is referred to as "protected state." (Protected state corresponds to the intuitive notion of "abstract state" that was used in the informal presentation.) A field, `f`, that is reachable from the object v is in v's protected state if `f` is not declared `assignable` and the object to which `f` belongs can only be reached from v through series of fields where each field resolves to have a read-only type when v is referred to through a read-only reference (see figure 9).

THEOREM 6. *A read-only reference,* `x`*, to object* v *cannot be used to reassign any field,* `f`*, within the protected state of* v.

PROOF. If `f` is declared `assignable`, then it is not in the protected state. If `f` is declared `final`, then it can never be

reassigned by rules RT-SET and RT-SET-THIS. Thus, the rest of the proof examines the case that `f` is declared to be this-assignable.

Without loss of generality, let v' be the object of which `f` is a field. By the definition of protected state, v' can only be reached from `x` through a reference that resolves to be read-only. Therefore, through all paths from `x`, `f`'s assignability resolves to final (see figure 9). The rules RT-SET and RT-SET-THIS show that `f`, resolving to final, cannot be reassigned through the reference `x`.

Now, one must examine the case that `x` or a field on the path between `x` and v' is assigned to a reference creating a aliasing, mutable reference to v'. If one could create such a reference, it could be used to reassign `f`. We proceed by case analysis on the reduction rules. The relevant cases are R-SET, R-INVK, R-NEW.

R-Set RT-SET allows references that resolve to be read-only to be assigned only to fields declared `readonly`; thus, `x` and all the fields between `x` and v' can only be assigned to read-only references by this rule.

RT-SET-THIS allows a this-mutable field, $\mathtt{f}_j$, which refers to object v'', to be assigned to another this-mutable field, $\mathtt{f}_i$, if $\mathtt{f}_j$ and $\mathtt{f}_i$ are fields of the same object. Suppose $\mathtt{f}_j$ and, therefore, v'' lie on the path from `x` to `f`. If a mutable reference to v'' through $\mathtt{f}_i$ is obtained by assigning $\mathtt{f}_j$ to $\mathtt{f}_i$, it could be used to reassign `f`. We show, by contradiction, that there cannot exist a mutable reference to v'' after the assignment. Being from the same object, $\mathtt{f}_i$ and $\mathtt{f}_j$'s mutability will always be identically resolved through any given reference. Thus, if there exists a mutable reference to v'' from `x` through $\mathtt{f}_i$ after the assignment, there must have already existed a mutable reference to v'' from `x` through $\mathtt{f}_j$ before the assignment. However, the prior existence of a mutable reference to v'' from `x` through $\mathtt{f}_j$ contradicts the fact that `f` is in the protected state of v.

R-Invk, R-New RT-INVK and RT-NEW allow references that resolve to be read-only only to be assigned to `readonly` formal parameters or fields, respectively. □

5. Templatizing methods over mutability to avoid code duplication

In Java, each (non-generic) `class` definition defines exactly one type. By contrast, in Javari, `class C { ... }` creates *two* types: `C` and `readonly C`. `C` contains some methods that are absent from `readonly C`, and a given method may have different signatures in the two classes (even though the method's implementation is otherwise identical).

Javari permits a programmer to specify the two distinct types using a single `class` definition, without any duplication of methods. (The alternative, code duplication, is unacceptable.)

Javari provides the keyword `romaybe` to declare that a method should be templated over the mutability of one or more formal parameters. If the type modifier `romaybe` is applied to any formal parameter (including `this`), then the type checker conceptually duplicates the method, creating two versions of it. (As noted earlier, it is not necessary for two versions of a class or method to exist at run time.) In the first version of the method, all instances of `romaybe` are replaced by `readonly`. In the second version, all instances of `romaybe` are removed. For example, the following code defines a `DateCell` class:

```
class DateCell {
  Date value;
  void setValue(Date d) /*mutable*/ { value = d; }
  romaybe Date getValue() romaybe { return value; }
  static romaybe Date cellDate(romaybe DateCell c) {
    return c.getValue();
  }
}
```

Its effect is the same as writing the following (syntactically illegal) code:

```
class readonly DateCell {
  Date value;
  readonly Date getValue() readonly {
    return value;
  }
  static readonly Date
  cellDate(readonly DateCell c) {
    return c.getValue();
  }
}

class DateCell extends readonly DateCell {
  Date value;
  void setValue(Date d) { value = d; }
  /*mutable*/ Date getValue() /*mutable*/ {
    return value;
  }
  static /*mutable*/ Date
  cellDate(/*mutable*/ DateCell c) {
    return c.getValue();
  }
}
```

Note that `setValue` only appears in `DateCell`, not `readonly DateCell`, because `setValue` is a mutable method. Also note that `DateCell` cannot inherit `readonly DateCell`'s `getValue`: the signatures are different due to the type of `this`.

Only one mutability type parameter (`romaybe`) is needed. Within a templated method, the mutability of the method parameters, including `this`, cannot affect the behavior of the method because the method must be valid for both read-only and mutable types. However, templated parameters can affect the types of the method's references, including the method's return type. Thus, the only reason for a `romaybe` template is to ensure that the method has the most specific return type possible. Since a method has only one declared return type, only one mutability template parameter is needed.[3]

5.1 Template inference

As an alternative to explicitly specifying method templates, Javari could instead use type inference to create a version of a method with a read-only return type. Many languages, such as ML [26], use type inference to permit programmers to write few or no type annotations in their programs; this is especially important when the types are complicated to write. Javari could similarly infer method templates, reducing the number of template annotations (`romaybe`) in the code.

Lack of explicit immutability constraints would eliminate the documentation benefits of Javari, or would cause method signatures to describe what a method does rather than what

[3] A method that both takes an array as a parameter and returns an array is an exception to this rule because an array can have multiple mutability modifiers. We feel that this case occurs too rarely in practice to warrant more complex templating syntax.

it is intended to do. Furthermore, programming environments would need to re-implement the inference, in order to present the inferred types to users.

Despite these problems, there are countervailing advantages to inference. For example, although we have not found it onerous in our experience so far, it is a concern that many methods and arguments would be marked as read-only, cluttering the code. (Were backward compatibility not an issue, we would have chosen different defaults for our keywords.) In future work, we plan to implement such an inference and determine whether users find it helpful.

A separate type inference issue is the need to build an inference system for existing Java libraries. Unannotated libraries use non-read-only types. This renders them essentially un-usable by Javari code: the type system prevents read-only references from being passed to libraries (lest they be modified by the library code). Programmers must be able to convert unannotated Java libraries (including those available only in compiled form) into versions with `readonly` annotations. This is an implementation issue rather than a language design issue, so we do not discuss it further in this paper.

6. Code outside the type system

Certain Java constructs, such as reflection and serialization, create objects in a way that is not checked by the Java type-checker, but must be verified at run time. We discuss how to integrate checking of mutability with these mechanisms, even though mutability has no run-time representation.

6.1 Reflection

Reflection enables calling a method whose return type (including mutability) is unknown at compile time. This prevents checking immutability constraints at compile time. We desire to maintain type soundness (reflective calls that should return a `readonly` reference must do so) and flexibility (reflective calls that should return a mutable reference can do so).

In particular, consider the `invoke` method:

```
package java.lang.reflect;
class Method {
  // existing method in Java (and Javari):
  /*mutable*/ Object invoke(java.lang.Object,
                            java.lang.Object...);
  // new method in Javari:
  readonly Object invokeReadonly(java.lang.Object,
                                 java.lang.Object...);
}
```

The three dots at the end of the parameter lists are not an ellipsis indicating elided code, but the Java 5 syntax for variable-argument routines.

Javari requires programmers to rewrite some uses of `invoke` into `invokeReadonly`, where `invokeReadonly` returns a `readonly Object` rather than an `Object` as `Method.invoke` does. For example:

```
Method m1 = ...;
Method m2 = ...;
Date d1 = (Date) m1.invoke(...);
readonly Date d2 =
  (readonly Date) m2.invokeReadonly(...);
```

`invokeReadonly` returns a read-only reference and does no special run-time checking. `invoke` returns a mutable reference, but performs a run-time check to ensure that the return type of the method being called is non-read-only. Note that this is a check of the invoked method's signature, *not* a check of the object or reference being returned. This checking is local and fast, unlike the immutability checking that is required to support general downcasts (section 7.5). To enable this check, the JVM can record the mutability of the return type of each method as the method is loaded.

This proposal takes advantage of the fact that the type-checker knows the compile-time types of the arguments to `invoke`. That is, in a call `foo(d)`, it knows whether the declared type of `d` is read-only. That information is necessary for resolving overloading: determining whether `foo(d)` is a call to `foo(Date)` or to `foo(readonly Date)` when both exist.

6.2 Serialization

Like reflection, serialization creates objects in a way that is outside the purview of the Java type system, and we wish to guarantee that code cannot use serialization to violate immutability constraints. Javari's serialized form includes a bit indicating whether the serialized object was referenced through a read-only reference. A bit is only needed for the top-level object that is deserialized (the argument to `writeObject`, which will become the `this` result of `readObject`).

There are two versions of `ObjectInputStream.readObject`, similarly to `Method.invoke`. The `readObjectReadonly` version returns a `readonly Object` and does no checking. The `readObject` version returns a mutable `Object`, but throws an exception if the read-only bit in the serialized representation is set.

7. Other language features

This section briefly explains several Javari features that have not changed from the Javari2004 dialect described in a previous paper and technical report [5, 4]. Full details can be found in those references.

7.1 Interoperability with Java

Javari is interoperable with Java and existing JVMs. The language treats any Java method as a Javari method with no immutability specification in the parameters (including `this`) or return type (and similarly for constructors, fields, and classes). Since the Javari type system does not know what a Java method can modify, it assumes that the method may modify anything.

While all Java methods can be called from Javari, Java code can only call Javari methods that does not use `readonly` in their return types. An implementation could enforce this by using standard Java names for non-read-only types, methods, and classes, and by "mangling" (at class loading time) the names of read-only types, methods, and classes into ones that cannot be referenced by legal Java code.

7.2 Type-based analyses

Javari enforces reference immutability — a read-only reference is never used to side-effect any object reachable from it. Reference immutability itself has many benefits. However, other guarantees may be desirable in certain situations. Four of these guarantees are object immutability (an object cannot be modified), thread non-interference (other threads

cannot modify an object), parameter non-mutation (an object that is passed as a `readonly` parameter is not modified), and return value non-mutation (an object returned as a `readonly` result is not modified). One advantage of reference immutability is that a subsequent type-based analysis (which assumes that the program type checks [30]) can often establish these other properties from it, but the converse is not true.

Extending reference immutability to stronger guarantees requires escape analysis or partial information about aliasing. Determining complete, accurate alias information remains beyond the state of the art; fortunately, the analyses do not require full alias analysis. Obtaining alias information about a particular reference can be easier and more precise than the general problem [2]. Programmers can use application knowledge about aliasing, new analyses as they become available, or other mechanisms for controlling or analyzing aliasing, such as ownership types [11, 3, 7], alias types [3], linear types [39, 14], or checkers of pointer properties [12, 17].

7.3 Inner classes

Javari protects a read-only enclosing instance from being mutated through an inner class. Placing `readonly` immediately following the parameter list of a method of an inner class declares the receiver *and all enclosing instances* of the receiver to be read-only.

Inner class constructors have no receiver, but placing the keyword `readonly` immediately following the parameter list of an inner class constructor declares all enclosing instances to be read-only. Such a constructor may be called a read-only constructor, by analogy with "read-only method". It is important to note that the "read-only" in "read-only constructor" refers to the enclosing instance. Read-only constructors do not constrain the constructor's effects on the object being constructed, nor how the client uses the newly-constructed object.

It is a type error for a read-only method or constructor to change the state of the enclosing instance, which is read-only. Furthermore, a non-read-only method or constructor cannot be called through a read-only reference.

7.4 Exceptions

Javari prohibits read-only exceptions from being thrown. This restriction, which has so far caused no difficulty in practice, is caused by our desire for interoperability with the existing Java Virtual Machine, in which (mutable) `Throwable` is a supertype of every other `Throwable`. It is possible to modify Javari to lift the restriction on throwing read-only exceptions, but the result is complicated, introduces possibilities for error in the type system and the implementation, and provides little practical benefit.

7.5 Downcasts

Every non-trivial type system rejects some programs that are safe — they never perform an erroneous operation at run time — but whose safety proof is beyond the capabilities of the type system. Like Java itself, Javari allows such programs, but requires specific programmer annotations (downcasts); those annotations trigger Javari to insert run-time checks at modification points to guarantee that no unsafe operation is executed. Among other benefits, programmers need not code around the type system's constraints when they know their code to be correct, and interoperation with legacy libraries is eased. The alternatives — prohibiting all programs that cannot be proved safe, or running such programs without any safety guarantee — are unsatisfactory, and are also not in the spirit of Java.

If a program is written in the type-safe subset of Javari, then static type-checking suffices. For our purposes, the unsafe operation is the downcast, which converts a reference to a superclass into a reference to a subclass. (In Java but not in Javari, these can also appear implicitly in certain uses of arrays of references, for which Java's covariant array types prevent sound static type-checking.) Java inserts checks at each down-cast (and array store), and throws an exception if the down-cast fails.

Javari's syntax for downcasting from a read-only type to a mutable type is "`(mutable)`*expression*". Regular Java-style casts may not be used to convert from read-only to mutable types. Special downcast syntax highlights that the cast is not an ordinary Java one, and makes it easy to find such casts in the source code.

Downcasting from a read-only to a mutable type triggers the insertion of run-time checks, wherever a modification (an assignment) may be applied to a reference that has had `readonly` cast away. (In the worst case, every assignment in the program, including libraries, must be checked.) The run-time checks guarantee that even if a read-only reference flows into a mutable reference, it is impossible for modifications to occur through the mutable reference. Thus, Javari soundly maintains its guarantee that a read-only reference cannot be used, directly or indirectly, to modify its referent.

A previous paper [5] describes an efficient technique for checking these casts at run time. It associates a "readonly" Boolean with each reference (*not* with each object). The readonly Boolean is true for each non-read-only reference derived from a `readonly` reference as a result of a downcast. The readonly Boolean is set when `readonly` is cast away, is propagated by assignments, and is checked whenever a modification (i.e., a field update) is performed on a non-read-only reference.

The following example illustrates the behavior of run-time casts.

```
class Foo {
  Date d;
  void setD() /*mutable*/ {
    d = new Date();
  }
}

         Foo f1 = new Foo();
readonly Foo rf = f1;
         Foo f2 = (mutable) rf;

f1.d = new Date();  // OK
f2.d = new Date();  // run-time error
f1.setD();          // OK
f2.setD();      // run-time error: at the second line of setD
```

Just as most new Java 5 code contains few Java casts, we believe that well-written new Javari code will contain few mutability downcasts. In a previous experiment, there was approximately one cast per 1000 lines [5]. However, that experiment annotated existing code (without improving its design) and used an earlier dialect of the Javari language that lacks many features that make casts less necessary.

8. Related work

8.1 Javari2004

The Javari language presented in this paper is an evolutionary improvement of an earlier dialect [5], which we call "Javari2004".

Experience with 160,000 lines of Javari2004 code indicated that Javari2004 is an easy-to-use language that retains the flavor and style of Java while providing substantial benefits, including improved documentation, extended ability to reason about code, and detecting errors in well-tested code. However, the Javari2004 design is deficient in a number of ways.

1. Conflates notions of assignability and mutability
2. Incompatible with generic types
3. Inflexible multi-dimensional arrays
4. Extra-linguistic macro-expansion templates
5. No support for reflection and serialization
6. No formal type rules or type soundness proof

The current Javari language corrects these problems. The changes are significant but are relatively small from the point of view of a user: most uses of the language, and its overall character, remain the same.

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 (section 3.3.2) highlight the orthogonality of assignability and mutability, and increase the expressiveness of the language. See appendix A for examples of the use of `assignable` and `mutable`.

This paper provides a detailed treatment of generic classes that smoothly integrates reference immutability into 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.

Javari2004 also proposed that a generic class could declare whether a field whose type is a type parameter is a part of the object's abstract state. We have discovered that such a declaration makes no sense. For a field whose type is a type parameter to be a part of the object's abstract state, it must be this-mutable; however, such a field cannot be this-mutable (section 3.2). Javari also disallows type parameters be modified with the `mutable` keyword (section 3.4).

As with generic classes, Javari permits programmers to independently specify the mutability of each level of an array (section 3.5). By contrast, Javari2004's specification states: "`readonly int[][]` and `readonly (readonly int[])` are equivalent," forbidding creation of a read-only array of mutable items.

Javari2004 integrated the syntax for templating a method over mutability with the syntax for Java 5's generic types. 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 [4, 5]).

Furthermore, Javari2004 allows declaring a multiple mutability type parameters. As noted in section 5, only a single mutability type parameter is sufficient, 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.

Reflection and serialization create objects whose types are not known to the static type checker. In Javari2004 (as in many other proposed type systems for Java), reflection and serialization create loopholes in the type system. The current Javari language is sound with respect to reflection and serialization, by introducing quick, local checks that can be performed at run time. See section 6.

Javari2004's type-checking rules [5, 4] are stated, for the full Javari2004 language (including inner classes and other Java idiosyncrasies) in the semi-formal style of the Java Language Specification [19]. This formulation is natural for many Java programmers, but it unsatisfying to others. This paper formalizes a core calculus for the Javari language. It builds on Featherweight Generic Java [20], adding side effects and assignability and mutability type modifiers. By presenting the typing rules for a language that is stripped to the bare essentials, we have made it easier to grasp the key features of Javari. Equally importantly, the formalization enables a type soundness proof for the Javari type system.

8.2 Other immutability proposals

Many other researchers have noticed the need for a mechanism for specifying and checking immutability. This section discusses other proposals and how ours differs from them.

Similarly to Javari, JAC [21] has a `readonly` keyword indicating transitive immutability, an implicit type `readonly T` for every class and 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. Javari obeys the Java type rules, uses a type checker rather than a preprocessor, and integrates

immutability with type parameterization. Additionally, we have implemented Javari and evaluated its usability [5].

The above comments also explain why use of read-only interfaces in Java is not satisfactory for enforcing reference immutability. A programmer could define, for every class `C`, an interface `RO_C` that declares the readonly 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 readonly 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.

Skoglund and Wrigstad [34] 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 type parameterization gives a less ad hoc and more disciplined way to specify families of declarations.

The functional methods of Universes [28] are pure methods that are not allowed to modify anything (as opposed to merely not being allowed to modify the receiver object).

Pechtchanski and Sarkar [31] 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%. Javari permits both of the lifetimes and supplies deep reachability, which complements the shallow reachability provided by Java's `final` keyword.

Capabilities for sharing [10] 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.

Porat et al. [32] provide a type inference that determines (deep) immutability of fields and classes. (Foster et al. [16] 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. This is the only other implemented tool for immutability in Java besides ours, but the tool is not publicly available for comparison.

Effect systems [23, 37, 29] 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. Our system is finer-grained than typical effect systems, operates over references rather than values, and considers all state reachable from a reference.

Our focus in this paper is on imperative object-oriented languages. In such languages, fields are mutable by default. In our type system, when a type is read-only, the default is for each field to be immutable unless the user explicitly marks it as mutable. Functional languages such as ML [25] use a different policy: they default all fields to being immutable. OCaml [22] combines object-orientation with a `mutable` annotation on fields (for example, references are implemented as a one-field mutable record). However, without a notion of read-only types, users are forced to hide mutability via use of interfaces and subtyping, which is less flexible and expressive than our proposal.

A programming language automatically provides a sort of immutability constraint for parameters that are passed, or results that are returned, by value. Since the value is copied at the procedure call or return, the original copy cannot be modified by the implementation or client, respectively. Pass- and return-by-value is typically used for values that are small. Some programming languages, such as Pascal and Ada, permit variables to be explicitly annotated as in, out, or in/out parameters; this is an early and primitive form of compiler-enforced immutability annotation.

8.3 C++ `const`

C++'s `const` keyword is intended to aid in interfaces, not symbolic constants [36]. Our motivation is similar, but our notion of immutability, and our type system, differ from those of C++, thus avoiding the 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. An unchecked cast can remove `const` from a variable, as can (mis)use of type system weaknesses such as unions and varargs (unchecked variable-length procedure arguments).

C++ permits the contents of a read-only pointer to be modified: read-only methods protect only the local state of the enclosing object. To guarantee transitive non-mutability, an object must be held directly in a variable rather than in 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. Finally, C++ does not permit parameterization of code based on the immutability of a variable.

By contrast to C++, Javari is safe: its type system contains no loopholes, and its downcast is dynamically checked. Furthermore, it differs in providing guarantees of transitive immutability, and in not distinguishing references from objects themselves; these differences make Javari's type system more uniform and usable. Unlike C++, Javari permits mutability of any level of an array to be specified, and permits parameterization based on mutability of a variable. Javari also supports Java features that do not appear in C++, such as nested classes.

Most C++ experts advocate the use of `const` (for example, Meyers advises using `const` wherever possible [24]). 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 [27, 18, 33]. We are not aware of any empirical (or other) evaluations regarding the costs and benefits of immutability annotations. Java programmers seem eager for compiler-checked immutability constraints: as of March 2005, support for `const` is the second most popular Java request for enhancement. (See `http://bugs.sun.com/bugdatabase/top25_rfes.do`. The most popular request is "Provide documentation in Chinese.")

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. [16]) eliminates such manual effort. Even without a type inference, we 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.

Another criticism of C++'s `const` is that it can occasionally lead to code duplication, such as the two versions of `strchr` in the C++ standard library. Mutability templates (section 5) make the need for such duplication rare in Javari. Finally, the use of type casts (section 7.5) permits a programmer to soundly work around problems with annotating a large codebase or with code duplication.

8.4 Related analyses

Reference immutability can help to prevent an important class of problems, in a simple and intuitive way. However, it is no panacea. Other techniques can address some of these issues, and there are many software engineering challenges that reference immutability does not address. We mention just a sample of other techniques.

Boyland [9] observes that mutational representation exposure (in which external code can corrupt a data structure) and observational exposure (in which external code can observe an internal representation changing) are duals: in each case, modifications on one side of an abstraction boundary are observable on the other. Reference immutability does not address observational exposure. Boyland argues that a language extension should not solve one of these problems without also solving the other. However, the problems are arguably different, since the latter is a result of a client improperly retaining a reference "too long," and even a value returned from `size()` may become out of date if it is retained too long (though it will never become an invalid integer). Mechanisms for solving all representation exposure problems are less mature, and it may be valuable to solve some important problems without solving them all.

Ownership types [11, 3, 7] provide a compiler-checked mechanism for preventing aliasing to the internal state of an object. As noted previously, alias, escape, and ownership analyses can enhance reference immutability. However, they do not directly address issues of immutability, including those not associated with abstraction boundaries. Ownership type annotations such as `rep` describe whether a reference is part of the object's state, whereas mutability annotations such as `readonly` indicate whether it can be modified; each approach has its advantages, and it would be interesting to combine them. Fractional permissions [8] are another mechanism for helping to avoid representation exposure. Finally, a type system based on linear logic [39, 14] can prevent multiple uses of a value, which may be useful, for example, when preventing representation exposure through constructor arguments.

9. Conclusion

We have presented a type system that is capable of expression, compile-time verification, and run-time checking of reference immutability constraints. Reference immutability guarantees 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. We have provided a set of formal type rules for a core calculus that models the Javari language, and we have used it to prove type soundness for the 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 a closely related dialect [5], these design features provide evidence that the language design is effective and useful.

Because our initial experience with Javari has been so positive, we are implementing the current version of the language, including type inference, in order to obtain more experience. This will permit realistic evaluation of Javari's strengths and weaknesses.

Acknowledgments

Adrian Birka implemented the Javari2004 prototype compiler. We are grateful to Joshua Bloch, John Boyland, Gilad Bracha, Doug Lea, Sandra Loosemore, and Jeff Perkins for their comments on the Javari design. The anonymous referees made helpful comments on an earlier version of this paper. This work was supported in part by NSF grants CCR-0133580 and CCR-0234651, DARPA contract FA8750-04-2-0254, and gifts from IBM and Microsoft.

References

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, Massachusetts, 1986.

[2] Alex Aiken, Jeffrey S. Foster, John Kodumal, and Tachio Terauchi. Checking and inferring local non-aliasing. In *PLDI*, pages 129–140, June 2003.

[3] Jonathan Aldrich, Valentin Kostadinov, and Craig Chambers. Alias annotations for program understanding. In *OOPSLA*, pages 311–330, October 2002.

[4] Adrian Birka. 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.

[5] Adrian Birka and Michael D. Ernst. A practical type system and language for reference immutability. In *OOPSLA*, pages 35–49, October 2004.

[6] Joshua Bloch. *Effective Java Programming Language Guide*. Addison Wesley, Boston, MA, 2001.

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

[8] John Boyland. Checking interference with fractional permissions. In *SAS*, pages 55–72, June 11–13, 2003.

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

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

[11] David G. Clarke, James Noble, and John M. Potter. Simple ownership types for object containment. In *ECOOP*, pages 53–76, June 2001.

[12] David Evans. Static detection of dynamic memory errors. In *PLDI*, pages 44–53, May 1996.

[13] David Evans, John Guttag, James Horning, and Yang Meng Tan. LCLint: A tool for using specifications to check code. In *FSE*, pages 87–97, December 1994.

[14] Manuel Fähndrich and Robert DeLine. Adoption and focus: Practical linear types for imperative programming. In *PLDI*, pages 13–24, June 2002.

[15] Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. Classes and mixins. In *POPL*, pages 171–183, January 1998.

[16] Jeffrey S. Foster, Manuel Fähndrich, and Alexander Aiken. A theory of type qualifiers. In *PLDI*, pages 192–203, June 1999.

[17] Jeffrey S. Foster, Tachio Terauchi, and Alex Aiken. Flow-sensitive type qualifiers. In *PLDI*, pages 1–12, June 2002.

[18] John D. Gannon. An experimental evaluation of data type conventions. *CACM*, 20(8):584–595, August 1977.

[19] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. Addison Wesley, Boston, MA, second edition, 2000.

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

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

[22] Xavier Leroy. *The Objective Caml system, release 3.07*, September 29, 2003. with Damien Doligez, Jacques Garrigue, Didier Rémy and Jérôme Vouillon.

[23] John M. Lucassen and David K. Gifford. Polymorphic effect systems. In *POPL*, pages 47–57, January 1988.

[24] Scott Meyers. *Effective C++*. Addison-Wesley, second edition, 1997.

[25] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, 1990.

[26] Robin Milner, Mads Tofte, Robert Harper, and Dave MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.

[27] James H. Morris. Sniggering type checker experiment. Experiment at Xerox PARC, 1978. Personal communication, May 2004.

[28] P. Müller and A. Poetzsch-Heffter. Universes: A type system for alias and dependency control. Technical Report 279, Fernuniversität Hagen, 2001.

[29] F. Nielson and H. R. Nielson. Type and effect systems. In *Correct System Design*, number 1710 in LNCS, pages 114–136. Springer-Verlag, 1999.

[30] Jens Palsberg. Type-based analysis and applications. In *PASTE*, June 2001.

[31] Igor Pechtchanski and Vivek Sarkar. Immutability specification and its applications. In *Java Grande*, pages 202–211, November 2002.

[32] Sara Porat, Marina Biberstein, Larry Koved, and Bilba Mendelson. Automatic detection of immutable fields in Java. In *CASCON*, November 2000.

[33] Lutz Prechelt and Walter F. Tichy. A controlled experiment to assess the benefits of procedure argument type checking. *IEEE TSE*, 24(4):302–312, April 1998.

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

[35] Daniel Dominic Sleator and Robert Endre Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, 32(3):652–686, July 1985.

[36] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, Boston, MA, special edition, 2000.

[37] Jean-Pierre Talpin and Pierre Jouvelot. The type and effect discipline. In *LICS*, pages 162–173, June 1992.

[38] Matthew S. Tschantz and Michael D. Ernst. Javari: Adding reference immutability to Java. Technical report, MIT Lab for Computer Science, August 2005.

[39] Philip Wadler. Linear types can change the world! In *IFIP TC 2 Working Conference on Programming Concepts and Methods*, pages 347–359, Sea of Galilee, Israel, April 1990.

A. Assignability and mutability examples

As shown in figure 2, each instance field can be declared with one of three assignabilities (assignable, unassignable, or this-assignable) and also with one of three mutabilities (mutable, read-only, and this-mutable). This appendix illustrates the use of Javari's reference immutability system through examples of all nine possibilities. As with the rest of this paper, we omit most visibility modifiers (`public`, `private`, ...) for brevity. Further-

more, for brevity this appendix does not explicitly address uses of type parameters.

A.1 this-assignable, this-mutable

This is the standard type for a field in a possibly-mutable class. All fields not declared `final` in Java code are interpreted as this type.

Suppose there is a class `Wheel` that is mutable (its pressure can change) and a class `Bicycle` that contains two `Wheel`s that may be changed (different wheels for different terrains).

```
class Wheel {
  int pressure;
}

class Bicycle {
  Wheel frontWheel;
  ...
}
```

A read-only reference to a `Bicycle` cannot be used to modify the `Bicycle` by reassigning `frontWheel` or mutating `frontWheel` by reassigning `pressure`. A mutable reference can modify `Bicycle` by reassigning `frontWheel` or changing its pressure.

A.2 `final`, this-mutable

Consider a file abstraction that contains a `StringBuffer` holding the contents of the file.

```
class MyFile {
  final StringBuffer contents;
}
```

The contents of a read-only file cannot be changed. `contents` should not be reassigned, and `StringBuffer` operations can be used to alter non-read-only files' contents as needed.

A.3 `assignable`, this-mutable

`assignable` fields can be used for caching. In the case of an `assignable` this-mutable field, it is caching an object that is this-mutable. For example, recall the `Bicycle` example from above. Suppose one wished to provide a method that returned the wheel with the greatest aerodynamic drag. If this method was costly to execute, one would wish to cache the result until one of the `Wheel`s changed.

```
class Bicycle {
  Wheel frontWheel;
  Wheel backWheel;

  private assignable Wheel mostDrag;
  romaybe Wheel mostDrag() romaybe {
    if (mostDrag == null || /*wheels have changed*/) {
      mostDrag = ...;
    }
    return mostDrag;
  }
}
```

Even when the reference to the `Bicycle` is read-only, `mostDrag` can be assigned with the result of the method. The cache and what it is caching — one of the `Wheel`s — are this-mutable.

A.4 this-assignable, `readonly`

Consider a class, `ChessPiece`, that represents a chess piece including the piece's position on a board. The position field, `pos`, should be modifiable for mutable references but not for read-only references. If the programmer wishes to reassign `pos`, instead of mutating the object assigned to the field, each time the piece is moved, then the field should be declared to be this-assignable and read-only.

```
class ChessPiece {
  readonly Position pos;

  readonly Position getPosition() readonly {
    return pos;
  }

  void setPosition(readonly Position pos) {
    this.pos = pos;
  }
}
```

A.5 `final`, `readonly`

`final readonly` fields are useful for state that should never change, including constants. A `ChessPiece`'s color should never change.

```
class ChessPiece {
  final readonly Color color;
}
```

A.6 `assignable`, `readonly`

Suppose that `ChessPiece` has a costly method, `bestMove`, that returns the best possible move for that piece. Since the method is costly, one would like the method to cache its result in case it is called again (before anything on the board has altered). Calling the `bestMove` method does not change the abstract state of the class, so it should be a `readonly` method. However, to allow the `bestMove` method to assign to the field that caches its result, the field must be declared `assignable`. Furthermore, since there is never a reason to mutate the position calculated, the field should be declared `readonly` to avoid programmer error.

```
class ChessPiece {
  private assignable readonly Position bestMove;
  readonly Position bestMove() readonly {
    if ((bestMove == null) || /* board changed */) {
      bestMove = ...;
    }
    return bestMove;
  }
}
```

A.7 this-assignable, `mutable`

Suppose one wishes to represent a set by an array and, for efficiency, move the last successfully queried item to the beginning of the array. The author must declare the array to be `mutable` to allow moving the last successfully queried item to the beginning of the array even when the set is read-only. The array must be declared this-assignable to allow reassigning the array when it reaches its capacity due to calls to `addElm`.

```
class MoveToFrontSet {
  private mutable Object[] elms;
  private int size;
  boolean contains(Object obj) readonly {
    for (int i = 0; i < size; i++) {
      if (elms[i].equals(obj)) {
        // should also check for null
        moveToFront(elms, i);
        return true;
      }
    }
    return false;
  }

  // Be sure to not declare method readonly
  void add(Object elm) {
    if (elms.length == size) {
      Object[] tmp = new Object[2*elms.length];
      for (int i = 0; i < elms.length; i++) {
        tmp[i] = elms[i];
      }
      elms = tmp;
    }
    elms[size] = elm;
    size++;
  }
```

```
  // Be sure to return readonly Object[]
  readonly Object[] toArray() readonly {
    return elms;
  }
}
```

The order that the elements appear in `elms` is not a part of the abstract state of the `Set` object; however, the fact that they are contained by the array assigned to `elms` is a part of the abstract state. This relationship is too complicated for the type system to capture, so the field must be declared `mutable`.

`elms` must be `mutable` so that the elements can be rearranged even when the instance of `Set` is read-only. Unfortunately, methods that add or delete elements could be declared `readonly` and still type check. Therefore, when writing code such as this, the programmer must be careful not to declare those methods read-only and to ensure that a mutable reference to `elms` does not escape.

A.8 `final`, `mutable`

Suppose one wishes to monitor the users who access a file. A simple way to do this is to require a user ID to be passed as an argument to the file's accessors and then record the ID in a set.

```
class File {
  final mutable Set<UserID> accessedFile;
  StringBuffer contents;

  readonly StringBuffer
  getContents(UserID id) readonly {
    accessed.add(id);
    return contents;
  }
}
```

The set `accessedFile` must be mutable so that a users ID may be added to it within the read-only method `getContents`.

A.9 `assignable`, `mutable`

Consider an implementation of a splay tree [35]:

```
class SplayTree<T extends readonly Comparable> {

  // The internal representation of a splay tree
  assignable mutable BinarySearchTreeNode<T> root;

  // Adjusts the tree so that newRoot becomes the root.
  splay(BinarySearchTreeNode newRoot) { ... }

  void insert(T val) { root.insert(val); }
  void delete(T val) { root.delete(val); }

  boolean find(T val) readonly {
    BinarySearchTreeNode node = root.find(val);
    if (node == null) {
      return false;
    } else {
      splay(node);
      return true;
    }
  }
}
```

In this example `assignable` and `mutable` are used because the type system is unable to capture how the abstract state of the class relates to its data structure at the field `root`. Without the `assignable` keyword, the root of the tree could not be reassigned by the `splay` method. The `mutable` keyword is also needed because the `splay` method needs to mutate the nodes rooted at `root` while rearranging the nodes within the tree.

B. Annotations

It is impractical to implement reference immutability using annotations instead of direct changes to the syntax of the Java language. This section discusses the issues involved. We believe that Javari should either use all keywords or all annotations; it would be confusing, and would offer little benefit, to mix the two.

Use of annotations is attractive. It ensures that Javari code is valid Java code. This guarantees interoperability with existing tools: Javari code can be compiled using any Java compiler, then run either by a JVM whose byte code verifier checks the immutability types encoded in the annotations, or by any JVM (losing the benefits of immutability checking). The Javari system could additionally work as a stand-alone type-checker. There would be no worries about breaking existing code that uses Javari keywords as identifiers. Avoiding changes to the programming language could encourage programmers to adopt Javari.

Unfortunately, the current Java annotation system is too weak to use to implement a reference immutability system. There are two main problems.

1. Annotations can only be applied to type declarations, not to other uses of types.
   (a) Annotations cannot be applied to a cast.
   (b) Annotations cannot be applied to the receiver (`this`) of a method. This could be worked around with a new annotation such as `@rothis`, which is syntactically applied to the method return type but semantically applies to the receiver type.
   (c) Annotations cannot be inserted at arbitrary locations within arrays. To express "`(readonly Date[])[][]`", one would need to write something like the (unintuitive) annotation `@readonly(2) Date[][][]` where the integer argument to the annotation indicates at what level the read-only modifier should be applied.
   (d) Annotations are not permitted on type parameters, so expressing this type would be difficult: `Map<List<readonly Date>, readonly Set<Number>>`.
2. Annotations on local variables are not recorded within the classfile by the `javac` compiler. Therefore, if we wish to use annotations and perform type checking on classfiles, we would be required to extend the annotation system. This would require changing the compiler, possibly by recording local variables' annotations within the local variable symbol table. A compiler change eliminates one of the benefits of using annotations: not requiring people to use a new compiler to check reference immutability constraints. (But the Javari code would remain backward-compatible with other Java compilers.)

A language change could still achieve backward compatibility with standard Java compilers by providing a special comment syntax where any comment that begins with "`/*=`" is considered as part of the code by the Javari compiler. (This approach was taken in LCLint [13], for example.) This feature allows the programmer to annotate an existing Java program with Javari's keywords without losing the ability to compile that program with a normal Java compiler. That comment mechanism could be supported by external tools, but should not be part of Javari proper.