A Practical Type System and Language for Reference Immutability

Adrian Birka Michael D. Ernst

MIT Computer Science and Artificial Intelligence Lab
Cambridge, MA 02139 USA
{adbirka,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 or objects 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.

In order to better understand the usability and efficacy of the type system, we have implemented an extension to Java, called Javari, that includes all the features of our type system. Javari is interoperable with Java and existing JVMs. It can be viewed as a proposal for the semantics of the Java `const` keyword, though Javari's syntax uses `readonly` instead. This paper describes the design and implementation of Javari, including the type-checking rules for the language. This paper also discusses experience with 160,000 lines of Javari code. Javari was easy to use and provided a number of benefits, including detecting errors in well-tested code.

Categories and Subject Descriptors

D.3.3 [**Programming languages**]: Language Constructs and Features—*data types*; F.3.1 [**Logics and meanings of programs**]: Specifying and Verifying and Reasoning about Programs; D.1.5 [**Programming techniques**]: Object-oriented programming

General Terms

Languages, theory, experimentation

Keywords

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

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
OOPSLA'04, Oct. 24-28, 2004, Vancouver, British Columbia, Canada.
Copyright 2004 ACM 1-58113-831-8/04/0010 ...$5.00.

1. Introduction

This paper presents a type system for specifying reference immutability: the transitively reachable state of the object to which a given reference refers cannot be modified using the reference. The transitively reachable state is the object and all state reachable from it by following references. A type system enforcing reference immutability has a number of benefits: it can increase expressiveness, enhance program understanding and reasoning by providing explicit, machine-checked documentation, save time by preventing and detecting errors that would otherwise be very difficult to track down, or enable analyses and transformations that depend on compiler-verified properties.

Our type system differs from previous proposals (for Java, C++, and other languages) in a number of ways. 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, or 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.

Our system offers guarantees for the entire transitively reachable state of an object. A programmer may use the type system to support reasoning about either the representation state of an object or its abstract state; in order to support the latter, parts of a class can be marked as not part of its abstract state. 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. Our type system permits excluding specific fields from the abstract state.

Our system 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. Our system also offers parameterization over immutability.

A type system is of limited interest if programmers cannot effectively use it. In the absence of experience using an implementation, the practicality of previous proposals is speculative. We have designed and implemented Javari (which stands for "Java with reference immutability"), an extension to the Java language that permits the specification and enforcement of reference immutability constraints. Javari specifies immutability constraints using the keyword `readonly`. The language is backward compatible with Java. In addition, Javari code is interoperable with legacy Java code, and runs on an unmodified Java Virtual Machine. The Javari compiler is publicly available at `http://pag.csail.mit.edu/javari/`.

We obtained experience with Javari by writing code in it, as well as by annotating Java code with `readonly` to convert it to Javari.

```
/** This class represents a set of integers. **/
public class IntSet {
  /** Integers in the set with no duplications. **/
  private int[] ints;

  /** Removes all elements from this that
   *  are not in set, without modifying set. **/
  public void intersect(IntSet set) {
    ...
  }

  /** Makes an IntSet initialized from an int[].
   * Throws BadArgumentException if there are
   * duplicate elements in the argument ints. **/
  public IntSet(int[] ints) {
    if (hasDuplicates(ints))
      throw new BadArgumentException();
    this.ints = ints;
  }

  /** Number of distinct elements of this. **/
  public int size() {
    return ints.length;
  }

  public int[] toArray() {
    return ints;
  }
}
```

Figure 1: A partial implementation of a set of integers.

In total, we have over 160,000 lines of Javari code, including the Javari compiler itself. This experience helped us design language features for Javari to make it more useful and easier to use. In addition, the experience helped clarify the benefits of using Javari.

This paper is organized as follows. Section 2 gives examples of the use of immutability constraints. Section 3 describes the Javari language, and section 4 presents our type-checking rules in the context of Javari. Section 5 relates our experience with using Javari. Finally, section 6 surveys related work, and section 7 concludes.

2. Examples

Reference immutability provides a variety of benefits in different situations. This section gives three simple examples of immutability constraints. The examples show enforcement of interface contracts, granting clients read-only access to internal data, and prevention of certain representation exposures. (Section 3.8 discusses how analyses can provide guarantees such as these by building on reference immutability, possibly with the assistance of a limited escape or alias analysis.) We use a class representing a set of integers (figure 1) to illustrate the problems and their solutions.

A method contract may state that the method does not modify some of its arguments, as is the case with `IntSet.intersect()`. Compiler enforcement of this contract guarantees that implementers do not inadvertently violate the contract and permits clients to depend on this property. Javari allows the designer of `IntSet` to write

```
public void intersect(readonly IntSet set) {
```

and the compiler verifies the method's specification about not modifying `set`.

Accessor methods often return data that already exists as part of the representation of the module. For example, consider the `toArray` method of the `IntSet` class. It is simple and efficient, but it exposes `IntSet`'s representation. A Java solution would be to return a copy of the array `ints` [5]. Our system permits a better solution:

```
public readonly int[] toArray() {
```

The `readonly` keyword ensures that the caller of `IntSet.toArray` cannot modify the returned array, thus permitting the simple and efficient implementation of the method to remain in place without exposing the representation to undesired changes.[1]

Representation exposure occurs when implementation details are accessible to clients. Java's access control mechanisms (for example, the `private` keyword) partly address this problem; Javari prevents some additional problems. A system for immutability can address the serious problem of *mutational* representation exposure, which permits modification of values and violation of data structure invariants, but is not relevant to *observational* representation exposure, which may be innocuous or desirable. For example, in the `IntSet` example of figure 1, the `toArray` accessor method exists to provide external access to the object's state.

In the `IntSet` example, the content of the private data member `ints` is externally accessible through the reference passed to the constructor `IntSet(int[])`. Client code can directly change the state of the `IntSet` object, which is undesirable. Even worse, client code can violate the representation invariant and put an `IntSet` object into an inconsistent state. For example, the client code could put a duplicate integer into the array `ints`, which would cause the method `IntSet.size()` to return an incorrect value.

Javari would catch this representation exposure at compile time. Since the constructor of `IntSet` is not intended to change the argument `ints`, the `IntSet` programmer would write

```
public IntSet(readonly int[] ints) {
```

and the compiler would issue an error at the attempt to assign `ints` to `this.ints`, preventing the `IntSet` programmer from forgetting to do a deep copy in the constructor.

3. The Javari language

The Javari language extends Java with explicit mechanisms for specifying immutability constraints, compile-time type checking to guarantee those constraints, and run-time checking for programs that use potentially unsafe casts.

Javari adds two new keywords to Java: `readonly` and `mutable`.[2] The `readonly` keyword specifies immutability constraints. The `mutable` keyword indicates a field that is not part of the abstract state of an object, or downcasts from read-only types to non-read-only types. The keywords are used as follows:

`readonly` is used in three ways:

1. As a type modifier: For every Java reference type `T`, `readonly T` is a valid type in Javari (and is a supertype of `T`), and a variable of such a type is known as a read-only reference.

[1] The returned array is aliased by the `ints` field, so `IntSet` code can still change it even if external code cannot. The (unspecified) semantics of `toArray` may permit this. The method specification might note that the returned reference reflects changes to the `IntSet`; alternately, the method specification, or an external analysis, might require that the result is used before the next modification of the `IntSet`.

[2] Java reserves, but does not currently use, the keyword `const`, so only one new keyword is strictly necessary. We have chosen to introduce the new keyword `readonly` instead of reusing `const` for two reasons. First, we believe that `readonly` better describes the concept of reference immutability. Second, we wish to avoid confusion with the C++ keyword `const` (see section 6.1).

Read-only references cannot be used to change the state of the object or array to which they refer. A read-only reference type can be used in a declaration of any variable, field, parameter, or method return type. A read-only reference type can also appear in a type-cast. See section 3.1.
2. As a method/constructor modifier: `readonly` can be used after the parameter list of a non-static method declaration (and likewise for an inner class constructor), making that method a read-only method. Read-only methods cannot change the state of the receiver ("`this`") object. Only read-only methods can be called through a read-only reference. See section 3.2.
3. As a class modifier: `readonly` can be used as a modifier in a class or interface declaration. It specifies that instances of that class or interface are immutable. See section 3.3.

`mutable` is used in two ways:

1. In a non-static field declaration, `mutable` specifies that the field is not part of the abstract state of the object. Mutable fields can be modified by read-only methods and through read-only references, while non-mutable fields cannot. See section 3.4.
2. In a type cast, `mutable` converts a read-only reference to a non-read-only reference. Run-time checks are inserted to maintain soundness and enforce immutability at run time. See section 3.6.

Javari supports type parameterization as a principled way to create related versions of code (section 3.5). 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. Also, Javari is interoperable with Java; Java and Javari code can call one another without recompilation (section 3.7). A postpass analysis can build on Javari's guarantee of reference immutability in order to make stronger guarantees about program behavior (section 3.8).

3.1 Read-only references

A read-only reference is a reference that cannot be used to modify the object to which it refers. A read-only reference to an object of type `T` has type `readonly T`. For example, suppose a variable `robuf` is declared as

```
readonly StringBuffer robuf;
```

Then `robuf` is a read-only reference to a `StringBuffer` object; it can be used only to perform actions on the `StringBuffer` object that do not modify it. For example, `robuf.charAt(0)` is valid, but `robuf.reverse()` causes a compile-time error.

When a return type of a method is a read-only reference, the code that calls the method cannot use the return value to modify the object to which that value refers.

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

The following rules for usage of read-only references (detailed in section 4) ensure that any code that only has access to read-only references to a given object cannot modify that object.

- A read-only reference cannot be copied, either through assignment or by parameter passing, to a non-read-only reference. In the `robuf` example above, a statement such as `StringBuffer buf = robuf;` would cause a compile-time error.
- If `a` is a read-only reference, and `b` is a field of an object referred to by `a`, then `a.b` cannot be assigned to and is a read-only reference.
- Only read-only methods (section 3.2) can be called on read-only references.

Javari also allows declarations of arrays of read-only references. For example, `(readonly StringBuffer)[]` means an array of read-only references to `StringBuffer` objects. For such an array, assignments into the array are allowed, while modifications of objects stored in the array are not. This is in contrast to `readonly StringBuffer[]`, which specifies a read-only reference to an array of `StringBuffer`s, and disallows both array element assignment and modification of objects stored in the array.

A non-read-only reference can be implicitly converted to a read-only one by an assignment, including implicit assignment to parameters during method or constructor invocations. A non-read-only reference can also be explicitly cast to a read-only one by using a typecast with a type of the form `(readonly T)`. For example:

```
readonly StringBuffer robuf = new StringBuffer();
StringBuffer buf = new StringBuffer();
robuf = buf;                              // OK
robuf = (readonly StringBuffer) buf;      // OK
buf = robuf;                              // error
buf = (StringBuffer) robuf;               // error
```

See section 3.6 for more details about type casts.

3.2 Read-only methods and constructors

Read-only methods are methods that can be called through read-only references. They are declared with the keyword `readonly` immediately following the parameter list of the method. It is a compile-time error for a read-only method to change the state of the receiver object. For example, an appropriate declaration for the `StringBuffer.charAt` method in Javari is:

```
public char charAt(int index) readonly
```

Read-only constructors are constructors that can be called with enclosing instance given through a read-only reference. (The Java Language Specification requires that a constructor for a (non-static) inner class be called on an instance of the outer class; this instance is called the enclosing instance.) Read-only constructors are declared with the keyword `readonly` immediately following the parameter list of the constructor. It is a compile-time error for a read-only constructor to change the state of the enclosing instance.

Read-only constructors can be used when the enclosing instance is supplied through a read-only reference, as they promise not to modify the enclosing object. Read-only constructors do not constrain the constructor's effects on the object being constructed, nor how the invoker uses the returned object.

Whether a method or constructor is read-only is part of its signature, and therefore it is possible to have two methods with the same name and parameters, if one is read-only and the other is not. Similarly, a read-only method declared in a subclass overloads (not overrides) a non-read-only one of same name and parameters declared in a superclass, and vice versa.

3.3 Immutable classes

A class or an interface can be declared to be immutable via the `readonly` modifier in its declaration.[3] This means that all of its

[3] We use the term "read-only type" to mean `readonly T`, for some `T`, and "immutable type" to mean a class or interface whose definition is marked with `readonly` and which therefore is immutable.

non-mutable non-static fields are implicitly read-only and final, and all of its non-static methods are implicitly read-only. In addition, if the class is an inner class, then all of its constructors are also implicitly read-only.

For an immutable class or interface `T`, read-only and non-read-only references to objects of type `T` are equivalent. In particular, read-only references can be copied to non-read-only references, something that is normally disallowed. Subclasses or subinterfaces of immutable classes and interfaces must be declared immutable. Any instance field inherited by an immutable class or interface must be either final and of read-only type, or mutable. Any instance method inherited by such a class or interface must be read-only.

3.4 Mutable fields

Mutable fields (declared with the `mutable` modifier) are not considered to be part of the abstract state of an object. A mutable field of an object O can be changed through a read-only reference to O.

One use of mutable fields is to enable creation of read-only containers that hold non-read-only elements. Another use of mutable fields is to cache results of read-only methods. For example, this situation arises in the Javari compiler, where a name resolution method `resolve()` needs to cache the result of its computation. The solution looks somewhat like the following:

```
class ASTName {
  ...
  private mutable Resolution res = null;
  public Resolution resolve() readonly {
    if (res == null)
      res = doResolve(); // OK: res is mutable
    return res;
  }
}
```

Without mutable fields, objects are unable to cache the results of read-only methods, and consequently Javari would force the programmer to either not label methods as read-only, or to take a significant performance penalty.

The `readonly`, `mutable`, and `final` keywords capture distinct concepts. If a field is declared as both `mutable` and `readonly`, the transitive state of the object it refers to is unchangeable through the field (because of `readonly`), while the field itself is assignable (because of `mutable`). If a field is declared as both `mutable` and `final`, it is not assignable (because of `final`), but the state of the object to which it refers is changeable through the field, even when the field is accessed through a read-only reference (because of `mutable`). If a field is declared with all three keywords, `mutable` has no effect.

3.5 Type parameterization

Javari adopts the type (class and interface) and method parameterization mechanisms of Java 1.5 [8], also known as “generics”. Among other benefits, this feature reduces code duplication. For example, it eases the definition of container classes, permitting one definition of `Vector` to be instantiated either as a container of `Objects` or of `readonly Objects`, and either as a container whose abstract state includes the elements or excludes the elements. More importantly, parameterization permits a single method definition to expand into multiple definitions that differ in the immutability of the parameters or of the method itself.

Javari augments the generics of Java 1.5 to permit a type qualifier (either `readonly` or the empty type qualifier) as a type parameter. This enables parameterization over whether a method is read-only (section 3.2). It also eases writing definitions in which references have different Java types but the same qualifier, such as this example from the `InvMap` class of Daikon (section 5.3):

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

Java compiles generics via erasure (after type checking is complete): the same bytecodes are used for all instantiations of the class or method. Our prototype compiler uses a different compilation strategy (for immutability generics only): it duplicates the parameterized declaration. Care must be taken when dealing with the different kinds of parameters — essentially, splitting them into possibly overlapping parts and processing the parts separately. (First, duplicate code as necessary to eliminate all immutability parameterization, and then process the remaining parameters in the usual way.) Likewise, when a cast changes both the Java type and whether the type is read-only (section 3.6), the cast is converted into two casts (one for the Java type and one for `readonly`), and each is processed separately.

3.6 Type casts

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 (including many uses of arrays), 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 typesafe 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. These can appear explicitly, and also in certain uses of arrays of references, for which Java's covariant array types prevent sound static type-checking. An example of the latter, where `B` is a subtype of `A` that defines `b_method`, is

```
B[] b_arr = new B[10];
A[] a_arr;
A a = new A();
a_arr = b_arr;
a_arr[0] = a;          // ArrayStoreException
b_arr[0].b_method();   // statically OK
```

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 non-read-only type is “`(mutable)` *expression*”. Regular Java-style casts cannot be used to convert from read-only to non-read-only 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. (See section 5.1.)

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

3.6.1 *Run-time checking*

Conceptually, Javari's checking works in the following way. The system associates a single "readonly" Boolean x_{ro} with each reference `x` (*not* with each object), regardless of the reference's static type. Array references are treated similarly, but they need both a Boolean for the array and a Boolean for each reference element. 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 `instanceof` operator checks the Boolean to determine whether `x instanceof mutable` for `x` of non-read-only static type.

The key rules for checking are as follows. Let `T` be a type, `a` and `b` be expressions, and `f` be a field name. First, each operation that yields a reference is modified to also produce a readonly Boolean for that reference. Two such rules are

$(a.f)_{ro} = a_{ro}$

$((\texttt{mutable})\ a)_{ro} = \text{true}$ where `a` is of read-only static type

Second, each field update is modified to check the readonly Boolean for the reference being updated. The statement `a.f = b;` becomes

```
if a_ro then throw exception;
a.f = b;
(a.f)_ro = b_ro;
```

Method calls are modeled as assignments from the actuals to the formals, plus an assignment from the return statement to the call result itself. The remainder of the rules are straightforward (for example, array elements are treated analogously to fields) and are omitted from this paper for brevity.

Many optimizations to the above rules are possible. For example, an analysis such as CHA [10] or RTA [3] could determine which classes an unsafe operation may be applied to, and which classes are reachable from those. Checks only need to be inserted in reachable classes. Furthermore, checks never need be inserted for immutable classes, nor for classes reached only via immutable references. (In other words, the more instances of `readonly` in the program, the less the overhead of run-time checking, if any, will be.) Checks can also be hoisted out of loops or eliminated where static analysis indicates them to be unnecessary. Explicit readonly Booleans need not be created for intermediate results.

We have implemented a prototype implementation of the run-time checking. It is inserted by a source-to-source postpass that is invoked if the program being compiled contains any unsafe casts or array uses. The resulting code then runs on an unmodified JVM. Our implementation is not highly tuned; for instance, it includes none of the above optimizations except for a partial hoisting optimization. Furthermore, because it is a source-to-source transformation, it incurs substantial overhead by introducing many wrapper classes to hold the readonly Boolean value that indicates whether a reference is immutable.[4] Despite all these inefficiencies, the prototype introduces slowdowns averaging less than 10% on our suite of real and benchmark programs containing no `readonly` annotations. (We ran our experiments using the Sun 1.4.1 JDK on a Pentium 4 running Red Hat Linux 7.2.)

[4] In some cases the Boolean can be added as a new variable or field — for example, if field `f` exists, then Boolean field f_{ro} is added and is manipulated or checked whenever `f` is used. In other cases, as for libraries for which source is not available, for arrays (which are not first-class objects), or for return values, wrapping is inevitable, introducing space and (especially) time costs.

Our main focus in this paper is the type system and its usability, which are more important than run-time efficiency, and we have not been hindered by these relatively small slowdowns. However, we believe the optimizations listed above could substantially reduce the overhead. Another approach is to modify the JVM directly, rather than operating at the source level. For instance, we could place the readonly Boolean in an unused bit in Java references. Such an approach would eliminate the need for new slots in objects and for wrapper objects. No updating code would be necessary for assignments: copying the reference would automatically copy the immutability Boolean. Checking code would still be required, and the bit would need to be masked out when pointers are dereferenced. Most seriously, a modified JVM would be required in order to run Javari programs. We hope to investigate such optimizations in future work.

3.7 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 or return type, or on the method (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.

This approach allows Javari to call Java code safely, without any immutability guarantees being violated. However, in many cases this analysis is over-conservative. For example, `Object.toString()` can safely be assumed to be a read-only method. Therefore, Javari permits the user to specify alternative signatures for methods, constructors, and fields in Java libraries or native code. The compiler trusts these annotations without checking them.

While Java methods can be called from Javari, Java code can only call Javari methods that do not contain `readonly` in their signatures. The Javari compiler achieves this by using standard Java names for non-read-only types, methods, and classes, and by "mangling" the names of read-only types, methods, and classes into ones that cannot be referenced by legal Java code. If a Javari program uses downcasts, which trigger the insertion of run-time checks (section 3.6), then any called Java code must be recompiled with the Javari compiler, or the run-time checks must be inserted at the call site, or the call must be proved to never pass a downcasted reference, or checks must be inserted by the JVM.

The Javari compiler guarantees that Javari code maintains the reference immutability constraints. However, a malicious client could subvert the type system by writing classes directly in JVM bytecode. In order to prevent such loopholes, the bytecode verifier should be extended to verify the reference immutability constraints, just as it verifies other type constraints that are initially checked at the source code level. Classfile checking suggests an alternative syntax: express information about reference immutability (in the source code and in classfiles) as metadata annotations, a new feature in Java 1.5. Such an approach would eliminate the need for the 1–2 new keywords in our proposal.

3.8 Type-based analyses

This section describes several type-based analyses that can be run after Javari type-checking in order to provide stronger guarantees. A type-based analysis assumes that the program type checks [27], but type-checking is independent of any subsequent analysis.

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 (section 5). However, other guarantees may be desirable in certain situations. Four of these guarantees are object immutability, thread non-interference,

parameter non-mutation, and return value non-mutation. One advantage of reference immutability is that a subsequent analysis can often establish these other properties from it (as outlined below), 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 below do not require full alias analysis. Obtaining alias information about a particular reference can be easier and more precise than the general problem [1]. Many programs use (some) pointers in disciplined and limited ways; these may correspond to, and may be motivated by, the places where programmers desire to reason about immutability. As new alias analyses become available, they can be applied to the below analyses; this is preferable to embedding assumptions about a particular alias analysis in the type system. Programmers can use other mechanisms for controlling or analyzing aliasing, such as ownership types [6], alias types [2], linear types [37, 13], or checkers of pointer properties [12, 15]. In the absence of an automated analysis, a programmer can use application knowledge about aliasing.

Object immutability. An object cannot be modified if all references to the object are read-only references. For example, this is guaranteed if at instantiation an object is assigned to a read-only reference, because only one reference exists. A further side condition is required: the constructor must not leak a non-read-only reference to the object or its parts. This circumstance is rare, and a trivial analysis reveals that almost no classes in the JDK leak even a read-only reference to `this`. The object immutability analysis accommodates interning, logging, tracing, and other mechanisms that retain/leak read-only references.

Another example is data structures that are treated differently by different phases of a program. For example, it is most natural for a graph or doubly-linked list to be mutable while being built, but it may be used immutably elsewhere. A simple analysis can often indicate when only immutable references remain after construction.

Thread non-interference. Other threads cannot modify an object if no other thread has a non-read-only reference to the object. Escape analysis for multithreaded programs [32] can indicate what references escape to other threads. In some cases the guarantee may be necessary only between synchronization points or within a critical region, which may be easier than a program-wide analysis.

Parameter non-mutation. An object that is passed as a `readonly` parameter is not modified until control exits the callee so long as no non-read-only reference aliases the object within the procedure scope (or in another thread; see above). Effectively, this means that the object should not be aliased by a non-read-only parameter or global variable. A variation on escape analysis [28, 9] can indicate whether an object may be placed in a non-read-only global variable. Parameter aliasing is simpler than the general alias problem, especially because parameters may be of incompatible types or may all be `readonly`.

Return value non-mutation. An object returned as a `readonly` result is not modified outside the callee's module so long as no non-read-only reference to the object escapes; this is a direct application of escape analysis.

3.9 Extension to new properties and languages

This paper presents a type system for reference immutability and develops it in the context of Java. While the intuition behind the enforced constraints is simple, our type rules and implementation cover the entire Java language. This permits us to explore how the type system interacts with real-world constructs including subtyping, inner types, exceptions, covariant arrays, and more. This experience leads us to believe that our approach should be easy to extend to other languages — especially imperative object-oriented languages, but most likely others as well. (The primary difficulty with a language such as C++ is its lack of type safety; C++'s loopholes prevent making any guarantees even after type checking.)

We also believe that our approach, which combines static type checking with dynamic checking where necessary and appropriate in order to keep the static rules simple and understandable by a programmer, should be extensible to other properties. For example, one interesting annotation would be `writeonly`, which permits specific clients to set a variable while prohibiting them from reading the value (for instance, seeing what others have written). A common paradigm that Javari supports (particularly when augmented by a type-based analysis) is an initialization phase followed by a read-only phase; this could be made explicit in a `writeonce` annotation that (like `readonly`) permitted unlimited reading. Such an analysis would be similar to a typestate analysis [34].

Further work is required to validate these intuitions and indicate the limits of our approach, but our success to date is encouraging.

4. Type-checking rules

Javari has the same run-time behavior as Java (except for possible run-time checks as described in section 3.6). However, at compile time, Javari ensures that modification of objects through read-only references, or similar violations of the language, do not occur. Section 4.1 introduces some definitions. Section 4.2 presents type-checking rules, in the style of the Java Language Specification [17]. A technical report [4] contains complete type checking rules. Section 4.3 gives intuitive explanations of our treatment of inner classes and exceptions.

4.1 Definitions

4.1.1 Javari's types

Javari's type hierarchy extends that of Java by including, for every Java reference type `T`, a new type `readonly T`. References of type `readonly T` are just like those of type `T`, but cannot be used to modify the object to which they refer.

Formally, the types in Javari are the following:

1. The null type `null`.
2. The Java primitive types.
3. Instance references. If `C` is any class or interface, then `C` is a type representing a reference to an instance of `C`.
4. Arrays. For any non-null type `T`, `T[]` is a type, representing an array of elements of type `T`.
5. Read-only types. For any non-null type `T`, `readonly T` is a type.

For convenience in the exposition later, we define the *depth* and the *base* of a type. Informally, the depth is just the nesting depth of an array type, while the base of an array type is the type with all array dimensions removed. Formally, for a type `T`, we define:

Depth:

- $depth(\texttt{T[]}) = depth(\texttt{T}) + 1$
- $depth(\texttt{readonly T}) = depth(\texttt{T})$
- $depth(\texttt{T}) = 0$ if `T` is null, primitive, or an instance reference

Base:

- $base(\texttt{T[]}) = base(\texttt{T})$
- $base(\texttt{readonly T}) = base(\texttt{T})$ if $base(\texttt{T})$ is a read-only type
- $base(\texttt{readonly T}) = \texttt{readonly}\ base(\texttt{T})$ if $base(\texttt{T})$ is a non-read-only type
- $base(\texttt{T}) = \texttt{T}$ if `T` is null, primitive, or an instance reference

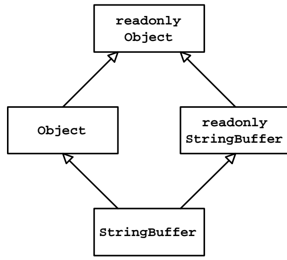

Figure 2: A portion of the Javari type hierarchy, which includes read-only and non-read-only versions of each Java reference type. Arrows connect subtypes to supertypes.

4.1.2 *Type equality and subtyping*

The equality relation on types is defined as follows:

1. For primitive types, the null type, and references to instances of classes and interfaces, two types are equal iff they are the same Java type.
2. `readonly T` and `readonly S` are equal iff $depth$(`T`) $=$ $depth$(`S`) and $base$(`readonly T`) $=$ $base$(`readonly S`).
3. `T[]` and `S[]` are equal iff `T` and `S` are equal.
4. For a non-read-only type `T`, `T` and `readonly S` are equal iff `T` and `S` are equal, and `T` is either primitive or is a reference to an instance of an immutable class or interface (section 3.3).

Item 2 implies that `readonly int[][]` and `readonly (readonly int[])[]` are equivalent. In other words, a read-only array of `int` arrays is the same as a read-only array of read-only `int` arrays. This is because read-only is a transitive property, and hence if a level of an array is marked as read-only, then all the lower levels are read-only.

Equal types are considered to be the same type. They are interchangeable in any Javari program.

A subtyping relationship (`T` subtype of `S`, written as `T` $<$ `S`) is also defined on types. It is the transitive reflexive closure of the following:

1. `byte` $<$ `char`, `byte` $<$ `short`, `char` $<$ `int`, `short` $<$ `int`, `int` $<$ `long`, `long` $<$ `float`, `float` $<$ `double`.
2. `null` $<$ `T` for any non-primitive type `T`.
3. If `T` and `S` are classes such that `T` extends `S` or interfaces such that `T` extends `S`, or `S` is an interface and `T` is a class implementing `S`, then `T` $<$ `S`.
4. For any non-null types `T` and `S`, if `T` $<$ `S` then `T[]` $<$ `S[]`.
5. For any non-read-only non-null type `T`, `T` $<$ `readonly T`.
6. For any non-read-only non-null types `T` and `S`, if `T` $<$ `S` then `readonly T` $<$ `readonly S`.
7. For any non-null type `T`, `T[]` $<$ `java.io.Serializable`, `T[]` $<$ `Cloneable`, and `T[]` $<$ `Object`.
8. For any non-read-only non-null type `T`, `(readonly T)[]` $<$ `readonly T[]`.

Figure 2 shows an example of the subtype hierarchy.

4.1.3 *Definitions relating to method invocations*

These definitions are the same as those in Java [17], except for the presence of the third clause in the definition of specificity. Our definitions and rules do not consider constructors to be methods; we always specify which (or both) we mean.

Compatibility: Given a method or constructor M and a list of arguments $A_1, A_2, \ldots A_n$, we say that the arguments are compatible with M if M is declared to take n parameters, and for each i from 1 to n, the type of A_i is a subtype of the declared type of the ith parameter of M.

Specificity: Given two methods of the same name or two constructors of the same class, M_1 and M_2, we say that M_1 is more specific than M_2 if the following three conditions hold:

1. M_1 and M_2 take the same number of parameters, say with types $P_1, P_2, \ldots P_n$ for M_1, and $Q_1, Q_2, \ldots Q_n$ for M_2, and for each i from 1 to n, P_i is a subtype of Q_i.
2. The class (respectively, interface) in which M_1 is declared is a subclass (subinterface) of the one where M_2 is declared, or M_1 and M_2 are declared in the same class (interface).
3. Either M_1 is not read-only or M_2 is read-only (or both).

4.2 Type-checking rules

4.2.1 *Programs*

A program type checks if every top-level class and interface declaration in the program type checks.

4.2.2 *Class/interface declarations*

A class or interface declaration type checks if all of the following hold:

1. One of the following two conditions holds:
   (a) The class or interface is immutable and each method inherited from any superclass or superinterface is static, or read-only, and each of the inherited fields inherited is static, mutable, or both final and of a read-only type.
   (b) The class or interface is not immutable, and neither is its direct superclass nor any of its direct superinterfaces.
2. No two fields of the same name are declared within the body of the class/interface.
3. No two methods of the same name and signature are declared within the body of the class/interface. The signature includes the number and declared types of the parameters, as well as whether the method is read-only.
4. No two constructors of the same signature are declared within the body of the class/interface.
5. Every field, method, constructor, member type, instance initializer, and static initializer declared within the class or interface type checks.

4.2.3 *Variable declarations*

For a field or local variable declaration of type T:

- If it does not have an initializer, it type checks.
- If it has an initializer of the form "`=` E" for an expression E, it type checks iff the assignment of the expression E to a left hand side with type `T` would type check.
- If it has an initializer of the form "`= {` $I_1, \ldots I_k$ `}` ", where I_k are initializers, it type checks iff `T = S[]` or `T = readonly S[]` for some type `S`, and the declaration `S v =` I_k or `readonly S v =` I_k respectively would type check for every k between 1 and n.

4.2.4 *Method declarations*

A method, constructor, instance initializer, or static initializer type checks if every expression, local variable declaration, and local type declaration in the body type checks.

4.2.5 *Expressions*

Each expression has a type and a boolean property called *assignability*. An expression is type checked recursively, with all subexpressions type checked first. If the subexpressions type check, then

their types and assignability are used to type check the given expression and deduce its type and assignability. Otherwise, the given expression does not type check.

For brevity, this paper gives only the type-checking rules for expressions that are substantially different from those of Java; a technical report [4] contains the complete type checking rules.

Assignment: The rules for type checking assignments are the same as in Java, except that the assignment expression does not type check if the lvalue is not assignable. The type of any assignment expression that type checks is the same as the type of the lvalue, and the expression is not assignable.

Typecast `(T)`A: In addition to the Java rules, a type cast must not cast from a read-only type to a non-read-only type. The type of the cast expression is `T` and the expression is not assignable.

Mutability typecast `(mutable)` A: Always type checks. If the type of A is non-read-only, this expression is of the same type. If the type of A is `readonly S[]` for some `S`, then the type of this expression is `(readonly S)[]`. If the type of A is `readonly S` where $depth$(`S`) $= 0$, the type of this expression is `S`. The expression is never assignable. This expression also triggers run-time checks (see section 3.6).

Receiver reference `this`: In a static context, `this` does not type check. In a non-static context, `this` has type `C` if `C` is a class and `this` appears inside a non-read-only method, a non-read-only constructor, or an initializer of `C`. Inside a read-only method or a read-only constructor of `C`, `this` has type `readonly C`. `this` is not assignable.

Containing object reference *name*`.this`: Type checks if it occurs in a non-static context in a method, constructor, or initializer of a class `I`, and *name* names a class `C` for which `I` is an inner class. The type of the expression is `C` unless it appears inside a read-only method or a read-only constructor of `I`, in which case the type is `readonly C`. This expression is not assignable.

Instance creation `new T(...)`: For creation of an inner class, if the enclosing reference is read-only, then only read-only constructors are eligible. If there is no enclosing reference, or the enclosing reference is not read-only, all constructors are eligible. The expression type checks if there is a most specific accessible eligible constructor compatible with the arguments to the class instance creation expression. The type of the expression is `readonly T` if the enclosing reference is a read-only reference, and `T` otherwise. This expression is never assignable.

Array dereference $A[E]$: Type checks if E is of integral type and A is of type `T[]` or `readonly T[]` for some type `T`; the type of the expression is respectively `T` or `readonly T`. The expression is assignable in the first case, and not assignable in the second.

Field access: Let `T` be the declared type of the field being accessed. If `T` is a read-only type, or the field is accessed through a non-read-only reference, or the field is a mutable or static field, then the type of the expression is `T`. Otherwise, the type of the expression is `readonly T`. The expression is assignable if the field is mutable or static, or if the reference through which the field is accessed is not a read-only reference. As an exceptional case, inside a read-only constructor, barename field access expressions for fields of the class that is being constructed are assignable.

Method invocation $E.m(\ldots)$: If the invoking reference is read-only, only static or read-only methods are eligible. If there is no invoking reference, only static methods are eligible. Otherwise, all methods are eligible. The expression type checks if there is a most specific accessible eligible method compatible with the arguments to the method invocation. The type of the expression is the declared return type of the method. A method invocation expression is never assignable.

Exception `throw` E: Javari prohibits read-only exceptions from being thrown. A technical report [4] describes other approaches that we rejected because they introduced loopholes into the type system or they require a complicated analysis that would provide very little benefit.

Special rule: Any reference to an instance of an immutable class `C`, when used inside the body of a non-immutable superclass or superinterface, is of type `readonly C`.

4.3 Inner classes and exceptions

Previous work on language-level immutability has not addressed inner classes or exceptions. This section explains how we handle them. It also provides an intuitive explanation of some of Javari's type checking rules.

4.3.1 Inner classes

The type-checking rules guarantee that read-only methods do not change any non-static non-mutable fields of `this`. The inner class feature of Java adds complexity to this guarantee. Javari must ensure that no code inside an inner class can violate an immutability constraint. There are three places in an inner class where immutability violations could occur: in a constructor, in a field or instance initializer, or in a method. The Javari type-checking rules (section 4) prevent any such violation. This section explains the rules by way of an example.

```
class Outer {
  int i = 1;
  // foo is readonly, so foo should not change i
  public void foo() readonly {
    class Local() {
      Local() readonly {
        i = 2;              // ERROR: changes i
        j = 3;              // OK
      }
      int j = ( i = 4 );  // ERROR: changes i
      void bar() readonly {
        i = 5;              // ERROR: changes i
      }
    }
    // Would be erroneous if either Local()
    // or bar() were not declared as readonly.
    new Local().bar();
  }
}
```

Constructors: Read-only constructors (see section 3.2) prevent this change. There are two possibilities for a change of `i` in a constructor of `Local`. This change could happen inside a read-only constructor, or inside a non-read-only one. If the constructor of `Local` is read-only, the assignment `i = 2` will not type check: read-only constructor bodies type check like read-only method bodies, except that a field of the class being constructed accessed by simple name is assignable and non-read-only. On the other hand, if the constructor of `Local()` is not read-only, it cannot be invoked, since Javari allows only read-only constructors to be invoked through a read-only enclosing instance, and the enclosing instance is implicitly `this`, a read-only reference inside `foo()`.

Initializers: If at least one read-only constructor exists in a given class or if an anonymous class is being constructed with read-only enclosing instance, the compiler treats instance initializers and instance field initializers as if they were in the body of a read-only constructor. The second case is necessary because anonymous constructors have an implicit constructor, which is considered a read-only constructor if the enclosing instance is read-only. This rule prevents modifications to the state of a read-only enclosing instance from initializers of inner classes.

Methods: The rule that `new Local()` must have type `readonly Local` if the enclosing instance is read-only prevents modification of the enclosing instance. If `bar()` is declared as read-only, the assignment to `i` inside it will fail to type check. If `bar()` is not declared as read-only, then the call to `bar()` in the example above does not type check, because `new Local()` has type `readonly Local`.

4.3.2 Exceptions

Our desire for interoperability with Java and the JVM complicates Javari's handling of exceptions. An exception thrown with a `throw` statement whose argument is a read-only reference should only be catchable by a `catch` statement whose parameter is declared as `readonly`, because otherwise the `catch` statement would be able to change the exception's state.

Javari prohibits read-only exceptions from being thrown. Therefore, the type-checker rejects some safe uses of read-only references to exceptions. This restriction has so far caused no difficulty in practice (see section 5).

A technical report [4] describes other approaches that we rejected because they introduced loopholes into the type system or they require a complicated analysis that would provide very little benefit. The key idea underlying these approaches is to wrap some exceptions at run-time in special wrapper objects, so that non-`readonly catch` statements do not catch read-only exceptions. Since Javari runs on an unmodified Java Virtual Machine and interoperates with legacy bytecode files, each wrapper class should be a subtype of `Throwable` (which Java treats specially). However, in Javari, `readonly Throwable` is a supertype of `Throwable` (see figure 2). Reconciling these problems is possible, but it is complicated, introduces possibilities for error in the type system and the implementation, and provides little practical benefit.

5. Experience

This section presents our experience with using Javari. Our primary goal was to gain qualitative experience, and our key result is that the type system and language are practical, usable, and effective. Section 5.1 expands on our observations. Then, the remainder of this section provides supporting details about our experience. Section 5.2 describes the Javari programs. Section 5.3 discusses the annotation process. Section 5.4 categorizes every type-checking error that occurred while annotating two of the five codebases, with examples of errors in each category. Section 5.5 discusses our experience with generics.

5.1 Qualitative results

A type system should be easy to use and understand. The language design should not overly constrain programmers, and it should fit with the way they work. Because practicality was one of our goals, we evaluated our type system, language, and implementation by both writing code in Javari and also annotating existing Java code with `readonly`. Writing code in Javari provides experience with the language the way many programmers would use it. In addition, it permits greater flexibility in working around type-checking errors than working with existing code does, and it can be more beneficial than annotation of existing code, since it provides earlier indication of errors. On the other hand, annotation of existing code is more quantifiable, since it is possible to track the time spent annotating and the problems discovered in original code. Also, it permits evaluation of how Javari fits with the existing practice of code written by programmers who did not have immutability in mind while coding.

The benefits of reference immutability included improving documentation (and making it consistent with the implementation), enhancing our understanding of the programs, eliminating errors, excising convoluted code, and efficiency improvements (by eliminating copying).

We found writing code from scratch to be faster, easier, and more natural than annotating existing code. The effort of adding `readonly` annotations seemed to be subsumed by that of writing the types, and we felt the ability to specify reference immutability caused us to think more formally about interfaces and make fewer design and implementation mistakes. Annotating existing code was not difficult, particularly since the code could be annotated module by module, relying primarily on local reasoning.

The bulk of our annotation effort was consumed in reverse engineering (determining both what the code did and what it ought to do) and in debugging. We strove to minimize disruptive code changes, but found that some problems could not be worked around entirely within the type system. The Javari problems (section 5.4.2) were few in number (rarer than Java casts) and most could be worked around by use of downcasts, though six unsafe uses required rewriting.

Our experience led us to add language features for parameterization and for downcasts. The parameterization feature was not strictly necessary, but it did prevent significant code duplication. Overall, adding parameterization was an easy task, especially when it parameterized over the immutability of the receiver object or when it was added identically to all subclasses of a class, as happened for several class hierarchies. See section 5.5 for more details.

The downcast mechanism was necessary because of legacy libraries, the pre-existing design of the software, and facts beyond the type system. We conclude that a fully statically enforced immutability type system is likely to be of limited applicability in practice. Using an explicit keyword (`mutable`) for immutability downcasts was critical in preventing us from accidentally leaving a Java cast in place but forgetting to add `readonly` to its type. Such a cast, if permitted to affect immutability, would cause Javari to silently insert run-time checks, and the error would be discovered only at run time. Without such a feature, or an analysis to indicate which casts are immutability downcasts, we estimate that at least a third, and perhaps many more, of the annotation errors we made would not have been caught until run time.

Our experience with Javari also convinced us that reference immutability is often sufficient. We were rarely able to use immutable types (object immutability); it is possible that redesigning the programs from the ground up may have increased the applicability of object immutability, but doing so would have had a severe performance cost. By contrast, reference immutability enabled us to reason about interfaces and program phases. We found human reasoning effective; we had not implemented the type-based analyses of section 3.8, but we suspect they would have helped even more.

Our experience revealed some weaknesses of our type system. In complex systems, transitive immutability may be too strong a constraint. For example, all Swing containers have child and parent references, so transitive immutability of any reference implies that almost every container is unchangeable. Thus, few references to Swing containers can be annotated as read-only. Similarly, legacy libraries may need access to a portion of the transitive state of a read-only reference (for an example, see section 5.4.2). Finally, our type system does not support reflection. These problems can be solved by run-time checks, but they reduce Javari's ability to help programmers by detecting errors at compile-time.

	New code	Annotated Java code			
Program	`java .util`	Javari	`java .util`	Gizmo-ball	Daikon
Code size					
classes	42	191	29	118	781
methods	373	1049	331	671	7454
lines	2412	16089	7666	15480	121607
NCNB lines	1740	9826	2793	9276	81269
Opportunities					
`readonly`	1312	4615	1269	2357	31687
Annotations					
`readonly`	854	2235	724	520	10369
`mutable`	24	28	9	17	110
generics	105	66	94	13	717
downcast	3	3	14	6	85
`readonly` uses					
local	81	468	93	69	1668
field	16	54	9	0	83
parameter	152	211	180	69	2148
cast	19	377	29	27	1148
return type	154	88	85	24	459
method	197	462	114	212	2913
class	1	37	1	3	94
generics arg	234	538	213	116	1856
Code errors	N/A	N/A	N/A		
Documentation				2	1
Implementation				1	19
Bad Style				3	4
Javari problems					
Inflexibility				3	23
Reflection				1	2
Annotation errors					
Signature				11	124
Implementation				31	486
Library				3	18
Time (hh:mm)	N/A	N/A	N/A		
Signature				2:40	5:30
Implementation				4:30	7:35
Type check and fix errors				6:10	55:05

Figure 3: Programs written in Javari or converted from Java to Javari. The number of classes includes both classes and interfaces. The number of methods includes constructors. "NCNB lines" is the number of non-comment, non-blank lines. "Opportunities" counts the number of places where `readonly` could appear (not including primitive types or common immutable classes such as `String` and subclasses of `Number`, and counting each multi-dimensional array as only one opportunity). Section 5.4 explains the error categories. Section 5.3 explains the time categories. Errors and time were recorded for only two of the programs.

5.2 Javari programs

We wrote many of the container classes in the `java.util` package from scratch in Javari, both to gain experience with Javari and because Javari requires parameterized container classes.

We also annotated four existing Java codebases.

Javari's compiler was our first major annotation experience. The annotation was intermixed with changes to the language and bug fixes in the compiler.

Sun container classes. We complemented the parameterized container classes we had written from scratch by annotating additional classes from the Sun JDK 1.4.1 reference implementation. We found an instance of bad coding style (see section 5.4).

Gizmoball, an extensible pinball game, is the final project in a software engineering class at MIT (6.170 Laboratory in Software Engineering). It was designed and written in two months by a group of four people that included the first author of this paper, who wrote about a third of the code.

Daikon is a tool for dynamic detection of likely invariants in programs [11]. The annotation was performed, without any help, by a programmer who had no previous experience with Daikon (not even as a user). The annotation time included time to understand this large, complex system well enough to annotate it.

5.3 Annotation process

Our methodology for annotating Java programs with `readonly` proceeded in three stages. During the first stage of the annotation ("Signature" in the bottom section of figure 3), we read the documentation and the signatures of all public and protected methods in the program, and marked the parameters, return types, and methods themselves with `readonly`. For example, if the documentation for a method specified that the method does not modify a parameter, we marked it with `readonly`. The second stage of the annotation ("Implementation" in figure 3) was to annotate the signatures of private methods and the implementations of all methods. The third stage ("Type check and fix errors") involved running the compiler on the resulting program, and considering and correcting any type checking failures.

Our annotation experience represents a pessimistic upper bound on the time cost, for six reasons.

1. The programmer was not familiar with the code, so the time includes program understanding and error diagnosing.
2. Completed code was annotated, but our experience indicates that the effort is significantly smaller when writing new code, because programmers already make design decisions about immutability, and catching errors would have recouped part or all of the cost.
3. The Javari system was still under development and contained some limitations and bugs.
4. Recording every type-checking error message (732 in all) was time-consuming and distracting.
5. The code contained no generic types, which would have eased the process.
6. We had no mechanical assistance such as type inference.

Despite these problems, the annotation process was quick and represented a small fraction of the original development time (under 5% for Gizmoball and considerably less for Daikon; 1200 and 1800 lines per hour, respectively).

5.4 Error classifications

The Javari compiler issues an error for three general reasons. First, the code may be erroneous, or the documentation may incorrectly describe it. Second, the Javari type system may be too restrictive, issuing an error about code that does not type check but that would be safe if it were run. Third, the user may have made an error during annotation, which the compiler has discovered. Figure 3 classifies each type checking error encountered while annotating the Gizmoball and Daikon systems, which together are 137,000 lines of code (91,000 non-comment non-blank lines). The final annotated code passed the Javari type checker without errors. This section describes the categorization of figure 3 and gives examples of errors in each category. A technical report [4] describes all 59 of the code errors and Javari problems.

5.4.1 *Code errors*

Code errors are problems with the original Java program that were discovered during the annotation and type checking process. These errors had remained in the code despite extensive testing; for example, Daikon has a worldwide user base and an 8-hour-long regression test that runs nightly.

We subdivided code errors into three subcategories.

Documentation: This subcategory represents errors in the documentation of a class or a public or protected method, causing an incorrect annotation. For example, the documentation of Daikon's `process_sample` method did not document that the second parameter, `ValueTuple vt`, is mutated by side effect. We updated the documentation to indicate this fact.

Implementation: This subcategory represents bugs in the original code, found during the type checking of the annotated code. The Gizmoball error was a representation exposure caused by a method improperly returning a reference to private data. We fixed it by adding `readonly` to the return type of the misbehaving method. An example of a Daikon error was a method that sorted its input array before computing some statistics about the array. We fixed this error by rewriting the method to do an array copy first. It was obvious that some of the errors could lead to user-visible failures; however, we did not verify this for every error.

Bad Style: This subcategory represents type checking failures caused by bad coding style. The code that causes these errors does not, to our knowledge, cause run-time failures, but it could have easily been written in a better style that would not only allow the code to type check, but would also make the program easier to maintain and debug.

For example, Gizmoball recalculates the size of gizmos that are displayed to the screen during each `paint()` call. A better alternative, which would also type check under Javari's rules, would be to recalculate only when the window size changes.

Daikon's method `canCreateAndWrite(readonly File file)` tests whether `file` can be created by calling `file.createNewFile()` and `file.delete()`. These calls fail to type-check, since they modify the `File` (and the file system). It is cleaner, and adequate for the purpose, to test whether the directory containing the file is writable.

Another example of bad coding style that is caught by the type system is in `java.util.TreeMap`. This class has a private method `buildFromSorted` that takes an `Iterator` parameter; this `Iterator` iterates either over keys or over entries in the map, depending on the value of a different parameter. It would be preferable to have two separate code paths for the two different iterators. (The code authors feared that if they duplicated this particularly subtle implementation, then in the future a maintainer might change one copy but not the other [19].) In Javari there is no correct typing of the `Iterator` parameter. The early release Java 1.5 libraries use a raw rather than a parameterized type for that `Iterator`.

5.4.2 Javari problems

In order to guarantee safety, type systems tend to conservatively reject programs that *might* behave unsafely — i.e., that cannot be proved to behave safely.

Inflexibility of the language: This subcategory represents safe code rejected by the type system's conservative analysis. We worked around them by omitting `readonly` annotations, by adding downcasts, or by rewriting code.

Half of all the type system inflexibility problems (13 instances in Daikon) were due to code of the form

```
readonly A a = new A();
foo(new A[] { a });       // ERROR
```

The array created by `new A[]` is typed as `A[]`, so cannot contain a `readonly A`. The problem was fixed by rewriting the code as

```
readonly A a = new A();
readonly A[] as = { a };
foo(as);
```

Another way to prevent the problem would be to change the language to introduce a new syntax for array constructors:

```
foo(new readonly A[] { a });
```

As another example, consider the following Gizmoball code snippet:

```
public class BuildDriver {
  private JFrame jf;
  ...
  private void askForLoad() readonly {
    final JDialog jd = new JDialog(jf, true);
    // use dialog box jd only to get a filename
  }
}
```

The `JDialog` constructor (not shown) is declared to take a mutable `JFrame`, because `JDialog.getOwner()` returns the `JFrame`, and a caller of `getOwner` often needs to modify the result. Thus, the `new JDialog` expression in the above code does not type check. Method `askForLoad` never calls `jd.getOwner()`, nor does `jd` escape that method, but those facts are beyond the type checker's static analysis, and so the compiler rejects this safe code. We fixed the problem by adding downcasts, similarly to adding a cast to work around Java's type system.

Reflection: This subcategory represents uses of reflection in the original program. Analysis of reflection is beyond our type system, and other Java type systems of which we are aware. (Indeed, reflection already permits Java code to violate accessibility modifiers and modify the contents of a `String`.) The dynamic checks of section 3.6 can ensure safety in the presence of reflection.

5.4.3 Annotation errors

This category represents mistakes committed by us during the annotation process. Usually, these were due to unfamiliarity with the code or to poor documentation. The *Signature Misannotation* subcategory represents errors due to an incorrect annotation of a signature of a public or protected method during the first stage of the annotation. The *Implementation Misannotation* subcategory represents the errors caused by an incorrect annotation of the type of a private field, the signature of a private method, the type of a local variable, or a type used in a cast expression. The *Library Misannotation* subcategory represents the errors caused by an incorrect annotation of a library method, for example an AWT method. (As noted in section 3.7, Javari provides annotated Java libraries. The library annotation time is not included in figure 3, but we discovered some errors in the library annotations while annotating client code.)

5.5 Generics

Our annotation introduced a fairly large amount of genericity ("generics" in figure 3). Of the uses of genericity in Daikon, 24 (3%) are parameterized types and the other 97% are parameterized methods. 48% of the parameterization appears in two external libraries that are included in the Daikon distribution, and they caused additional genericity at uses of the libraries. Overall, for method parameterization, the type parameters were used in the following ways:

in return type	56%
in formal types	59%
in method type	79%
as type parameter	50%

We found it natural and flexible to declare the immutability of a reference at a variable declaration, rather than using an immutable

type (which Javari also supports). Furthermore, adding a type parameter would have required a change to all client code that mentioned the type name. As a minor point, method parameterization resulted in less code duplication. Perhaps some of the genericity could have been converted from method to type generics; this would further reduce the number of annotations required.

6. Related work

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 [18] 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.

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 by 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.

Skoglund and Wrigstad [33] 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 (adopted from Java 1.5) gives a less ad hoc and more disciplined way to specify families of declarations.

The functional methods of Universes [25] 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 [29] 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 [7] 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. [30] provide a type inference that determines (deep) immutability of fields and classes. (Foster et al. [14] 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 [21, 36, 26] 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 [23] use a different policy: they default all

fields to being immutable. OCaml [20] 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-vale 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.

6.1 C++ `const`

C++'s `const` keyword is intended to aid in interfaces, not symbolic constants [35]. 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++ provides no 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 [22]). 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 [24, 16, 31]. 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 May 2004, support for `const` is the fourth most popular Java request for enhancement.[5]

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. [14]) 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. Immutability parameters (section 3.5) make the need for such duplication rare in Javari. Finally, the use of type casts (section 3.6) permits a programmer to soundly work around problems with annotating a large codebase or with code duplication.

7. 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 language. We have implemented the language and presented experience with non-trivial Javari programs. The evidence suggests that, although a language designer's budget (in terms of new language features) is limited, reference immutability is worthy of serious consideration and further investigation.

Our goal is not to produce a complicated and subtle new type system, but a solution to an important problem that others have grappled with unsuccessfully. Many of the components of our approach have previously appeared in the literature. We have synthesized these pieces in a novel way, resulting in a simple and effective approach. This paper's contributions include the following.

We chose 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.

We proposed a set of type-based analyses that can run after type checking in order to make stronger guarantees or to enable verification or transformation. For example, we show how to guarantee object immutability. The type-based analyses require only limited aliasing information; usually a simple escape or alias analysis suffices, but an arbitrary alias analysis may be used.

We combined compile-time and run-time checking to create an effective, practical, and safe system. The system detects all violations of the immutability constraints at compile time, in the absence of immutability downcasts. If a programmer chooses to use downcasts (which are sometimes essential for interoperability with legacy code or to express application invariants), then efficient run-time checking at modification points catches all unsafe uses while permitting safe ones.

We provided a safe type system for transitive immutability in the context of a full, real, object-oriented language, Java, rather than a model or subset. It is simple to see how our ideas would apply to an idealized toy language, but it was nontrivial to support a real language with all its wrinkles. Our success indicates that the system is comprehensible and usable in practice.

The syntax and semantics of Javari are backward compatible

[5] See `http://developer.java.sun.com/developer/bugParade/top25rfes.html`. The first and third most popular requests (generics and covariant return types) are addressed by the Java 1.5 language, and the second most popular request is "Provide documentation in Chinese."

with Java and the Java Virtual Machine. Java and Javari code can call one another in a safe manner. This compatibility and interoperability with Java eases the transition between the languages; developers can continue to use existing libraries and can adopt a pay-as-you-go strategy to annotating their code with immutability constraints. Javari is also faithful to the spirit of Java: it feels like Java and introduces run-time checks only as a result of constructs that already result in Java run-time checks.

We have provided the first implementation and evaluation of transitive (deep) immutability in the context of a safe language. Experience with 160,000 lines of Javari code demonstrates that the syntax, rules, and checking are workable in practice. Even a user unfamiliar with a 120,000-line program was able to annotate it quickly and discover errors. Our study provides insight into what immutability-related errors users make in the absence of Javari's features.

Javari revealed over two dozen real errors in well-tested code. The benefits, especially in reducing wasted time and clarifying specifications, are potentially much greater when Javari is used throughout the development cycle rather than after the fact. Even the guarantee of reference immutability alone is enough to improve documentation, find errors, and expose other problems.

Acknowledgments

Iuliu Vasilescu implemented the prototype run-time checking system. We are grateful to Adam Kieżun, Chandra Boyapati, Craig Chambers, Walter Tichy, and the anonymous referees for their helpful comments on a draft of this paper. This work was supported in part by NSF grant CCR-0133580 and a gift from TIBCO Software.

References

[1] A. Aiken, J. S. Foster, J. Kodumal, and T. Terauchi. Checking and inferring local non-aliasing. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, pages 129–140, San Diego, CA, June 9–11, 2003.

[2] J. Aldrich, V. Kostadinov, and C. Chambers. Alias annotations for program understanding. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2002)*, pages 311–330, Seattle, WA, USA, Oct. 28–30, 2002.

[3] D. F. Bacon and P. F. Sweeney. Fast static analysis of C++ virtual function calls. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 324–341, San Jose, CA, USA, Oct. 6–10, 1996.

[4] A. Birka. Compiler-enforced immutability for the Java language. Technical Report MIT-LCS-TR-908, MIT Laboratory for Computer Science, Cambridge, MA, June 2003. Revision of Master's thesis.

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

[6] C. Boyapati, B. Liskov, and L. Shrira. Ownership types for object encapsulation. In *Proceedings of the 30th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 213–223, New Orleans, LA, Jan. 15–17, 2003.

[7] J. Boyland, J. Noble, and W. Retert. Capabilities for sharing: A generalisation of uniqueness and read-only. In *ECOOP 2001 — Object-Oriented Programming, 15th European Conference*, pages 2–27, Budapest, Hungary, June 18–22, 2001.

[8] G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '98)*, pages 183–200, Vancouver, BC, Canada, Oct. 20–22, 1998.

[9] J.-D. Choi, M. Gupta, M. J. Serrano, V. C. Sreedhar, and S. P. Midkiff. Escape analysis for Java. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '99)*, pages 1–19, Denver, Colorado, Nov. 3–5, 1999.

[10] J. Dean, D. Grove, and C. Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *ECOOP '95, the 9th European Conference on Object-Oriented Programming*, pages 77–101, Åarhus, Denmark, Aug. 7–11, 1995.

[11] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):1–25, Feb. 2001. A previous version appeared in *ICSE '99, Proceedings of the 21st International Conference on Software Engineering*, pages 213–224, Los Angeles, CA, USA, May 19–21, 1999.

[12] D. Evans. Static detection of dynamic memory errors. In *Proceedings of the SIGPLAN '96 Conference on Programming Language Design and Implementation*, pages 44–53, Philadelphia, PA, May 21–24, 1996.

[13] M. Fähndrich and R. DeLine. Adoption and focus: Practical linear types for imperative programming. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, pages 13–24, Berlin, Germany, June 17–19, 2002.

[14] J. S. Foster, M. Fähndrich, and A. Aiken. A theory of type qualifiers. In *Proceedings of the ACM SIGPLAN '99 Conference on Programming Language Design and Implementation*, pages 192–203, Atlanta, GA, May 1–4, 1999.

[15] J. S. Foster, T. Terauchi, and A. Aiken. Flow-sensitive type qualifiers. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, pages 1–12, Berlin, Germany, June 17–19, 2002.

[16] J. D. Gannon. An experimental evaluation of data type conventions. *Communications of the ACM*, 20(8):584–595, Aug. 1977.

[17] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification*. Addison Wesley, Boston, MA, second edition, 2000.

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

[19] D. Lea. Personal communication, Aug. 1, 2004.

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

[21] J. M. Lucassen and D. K. Gifford. Polymorphic effect systems. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pages 47–57, San Diego, CA, Jan. 1988.

[22] S. Meyers. *Effective C++*. Addison-Wesley, second edition, 1997.

[23] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, 1990.

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

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

[26] F. Nielson and H. R. Nielson. Type and effect systems. In E. R. Olderog and B. Steffen, editors, *Correct System Design*, number 1710 in Lecture Notes in Computer Science, pages 114–136. Springer-Verlag, 1999.

[27] J. Palsberg. Type-based analysis and applications. In *ACM SIGPLAN/SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'01)*, Snowbird, Utah, USA, June 18–19, 2001.

[28] Y. G. Park and B. Goldberg. Escape analysis on lists. In *Proceedings of the SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 116–127, San Francisco, CA, June 17–19, 1992.

[29] I. Pechtchanski and V. Sarkar. Immutability specification and its applications. In *Joint ACM-ISCOPE Java Grande Conference*, pages 202–211, Seattle, WA, Nov. 3–5, 2002.

[30] S. Porat, M. Biberstein, L. Koved, and B. Mendelson. Automatic detection of immutable fields in Java. In *CASCON*, Mississauga, Ontario, Canada, Nov. 13–16, 2000.

[31] L. Prechelt and W. F. Tichy. A controlled experiment to assess the benefits of procedure argument type checking. *IEEE Transactions on Software Engineering*, 24(4):302–312, Apr. 1998.

[32] A. Salcianu and M. Rinard. Pointer and escape analysis for multithreaded programs. In *Proceedings of the 8th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 12–23, Snowbird, Utah, June 18–20 2001.

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

[34] R. E. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Transactions on Software Engineering*, SE-12(1):157–171, Jan. 1986.

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

[36] J.-P. Talpin and P. Jouvelot. The type and effect discipline. In *Proceedings of the Seventh Annual IEEE Symposium on Logic in Computer Science*, pages 162–173, Santa Cruz, CA, June 22–25 1992.

[37] P. 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, Apr. 1990.