Tools for Enforcing and Inferring Reference Immutability in Java

Telmo Luis Correa Jr. Jaime Quinonez Michael D. Ernst
MIT Computer Science and Artificial Intelligence Lab, Cambridge, MA, USA
telmo@csail.mit.edu jaimeq@csail.mit.edu mernst@csail.mit.edu

Abstract

Accidental mutation is a major source of difficult-to-detect errors in object-oriented programs. We have built tools that detect and prevent such errors. The tools include a javac plug-in that enforces the Javari type system, and a type inference tool. The system is fully compatible with existing Java programs.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features—data types; F.3.1 [*Logics and Meaning of Programs*]: Specifying and Reasoning and Verifying about Programs; D.1.5 [*Programming Techniques*]: Object-oriented Programming
Keywords assignable, immutability, Java, Javari, mutable, readonly, side effects, type system, verification

1. Introduction

Accidental mutation errors are difficult to detect, since the unintended mutation itself is not different from other mutations that happen throughout the program, and a mutation error is not immediately detected at run time. We present a type-system based solution to the accidental mutation problem.

Javari [6] is an extension of the Java language that permits the specification and compile-time verification of immutability constraints. Programmers can state the mutability and assignability of references using a small set of type annotations. The extension is implemented through Java annotations, keeping the code backwards compatible.

We have built two related tools: the Javarifier, an implementation of a type inference algorithm for Javari, and the Javari type checker, a plug-in for the javac compiler that enforces the annotations related to immutability.

The Javarifier can be used to automatically annotate existing code, enabling Javari to be more easily adopted and preventing accidental mutability errors in future code that uses the annotated code. The Javari checker is a javac plug-in that enforces the Javari annotations, detecting incorrect mutations in a program.

The Javari language toolset can be downloaded from **http:/pag.csail.mit.edu/javari**.

OOPSLA'07, October 21–25, 2007, Montréal, Québec, Canada.
ACM 978-1-59593-786-5/07/0010.

2. Short overview of Javari

The Javari language [5, 6] was designed to fulfill the following goals:

Non-conversible: a readonly reference cannot be assigned to a mutable reference (after which it might be modified).
Transitive: the provided immutability is deep through fields, allowing reasoning about an object's abstract state.
Flexible: parts of an object's concrete state can be excluded from an object's abstract state.
Compatible: the language is backwards compatible with existent Java code.
Usable: the language should be intuitive to programmers.

Javari's type system has a number of differences from previous immutability proposals. Instead of object immutability, it offers reference immutability, which is more flexible: the same object may be referenced by read-only and mutable references, and can still provide guarantees about code that manipulates the readonly references. This permits, for example, returning a readonly reference to an existing object, instead of making a copy to preserve its original state. Javari's guarantee is transitive: no state may be modified when accessed through an immutable reference's fields. This permits a programmer to reason about objects and their abstract state.

Each reference in Javari has a mutability of **readonly** or **mutable**. Mutability determines whether a variable's value can be side-effected. By default, the fields of a **readonly** reference are treated as **readonly** and **final**. A **mutable** reference cannot be assigned to a **readonly** reference. Each reference also has an assignability; Java's **final** keyword makes a variable unassignable, and Javari's **assignable** keyword makes an otherwise **final** variable **assignable**.

The default values for mutability and assignability (**this-mutable** and **this-assignable**) are chosen for backward compatibility with existing code, inheriting the meaning from its enclosing elements. The keyword **? readonly** provides

a wildcard for mutability, while the keyword **romaybe** provides a behavior akin to mutability overloading or templates.

3. Implementation

ConstJava is an implementation of a previous reference immutability proposal that later evolved to the Javari2004 language [1]. Unlike ConstJava, our Javari implementation uses Java 1.5 generics and Java's standard syntax for casts, instead of explicit keywords for immutability polymorphism and downcasts.

Our implementation is the first for the current Javari language [6], and incorporates several improvements to the language design [5].

This implementation of Javari is based upon the annotation system proposed in JSR 308 [3, 4]. The implementation of the keywords as annotations (which may be enclosed in comments) ensures that the code is backwards compatible, while not affecting the runtime behavior of the program.

3.1 Experience

The Javarifier type inference has processed tens of thousands of Java code, rewriting it into type-correct Javari code by inserting annotations.

After one month of work, the Javari type checker was already able to check a hand-annotated version of the JOlden benchmarks [2]. JOlden is a Java version of a suite of pointer-intensive C programs. We have since continued to improve the toolset, which is ready for use by Java programmers.

4. Examples

The following examples [6] illustrate how Javari can be used to prevent errors in Java programs.

Consider the following routine in a voting system:

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

It is necessary to ensure that the input votes is not modified. Using Javari, the specification for this method could declare that the input is read-only.

```
ElectionResults tabulate(@ReadOnly Ballots votes) {
  ... // cannot tamper with votes
}
```

Accessor methods often return part of an object's internal representation. For example, in the JDK 1.1.1, the **Class.getSigners** method has an implementation similar to the following:

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

This represents a security hole, since a malicious client could call **getSigners** and then modify the array. Javari permits the following fix:

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

The **@ReadOnly** annotation ensures that the array returned by the method cannot be modified through the returned reference, in the caller or at any place that reference is passed to. Another possible solution, actually implemented in later versions of the JDK, is to return a copy of an object's internal state. Making a copy, however, can be a computationally expensive process. For example, a file system could grant a client read-only access to its clients:

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

Javari allows the programmer to avoid the cost of making a copy by declaring the return type of the method as:

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

As a last example, reference immutability can be used to ensure object immutability, if all references to an object are immutable. For example, there is only one reference to an object when it is first constructed. As another example, some objects may need to be treated as mutable only while being initialized, but should be immutable thereafter. Javari can be used to specify those constraints:

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

References

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

[2] B. Cahoon and K. S. McKinley. Data flow analysis for software prefetching linked data structures in Java. In *PACT*, pages 280–291, Sept. 2001.

[3] M. D. Ernst and D. Coward. JSR 308: Annotations on Java types. `http://pag.csail.mit.edu/jsr308/`, Oct. 17, 2006.

[4] M. M. Papi and M. D. Ernst. Compile-time type-checking for custom type qualifiers in Java. In *OOPSLA Companion*, Oct. 2007.

[5] M. S. Tschantz. Javari: Adding reference immutability to Java. Master's thesis, MIT Dept. of EECS, Aug. 2006.

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