Putting Javari into Practice

Matt McCutchen

under the direction of
Dr. Michael Ernst
Program Analysis Group
Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology

Research Science Institute
August 1, 2006

Abstract

This paper presents a practical evaluation of the Javari programming language, a variant of Java in which one can prevent writing through object references. Javari adds a `@ReadOnly` type qualifier to Java and enforces appropriate type safety rules. The specification of the language has been given, and its implementation is in progress; in this paper, we suggest usability improvements to Javari and resolve some of the issues that Javari programmers will face when they begin to use the language.

1 Introduction

Type qualifiers have long been used in programming languages to provide compiler-enforceable guarantees about program safety. The most useful type qualifiers are probably those that prevent a program from modifying a variable, such as the `const` qualifier in C++. Declaring a variable to be `const` helps prevent some obvious programming errors, such as the accidental use of `=` instead of `==` in an `if` condition. Declaring the target of a pointer to be `const` has more interesting results: the program cannot modify the target object through *that pointer*. Nevertheless, a pointer to the same object that allows modification may be available elsewhere in the program.

One may convert a pointer that allows modification to one that does not before passing it to another part of the program, but there is no legal way to perform the opposite conversion. Thus, `const` qualifiers on pointer targets serve to restrict which parts of the program may modify an object. If they are applied properly, the compiler can signal bad programming practices that could result in unexpected modifications as type safety errors.

Without a qualifier like `const`, it is easy for a programmer working on one section of a large piece of unfamiliar software to introduce a bug by modifying an object at the wrong time. These bugs tend to be very difficult to track down. A `const` qualifier encourages programmers to think clearly about which objects should be modified where and to express assumptions that objects will not be modified in the source code so that others will not inadvertently violate them.

The Java language [6] uses the `final` qualifier to prevent assigning to variables but currently lacks a qualifier to prevent modifying the target of a reference. The earliest attempts to add such a qualifier to Java were the Java with Access Control (JAC) language described by Kniesel and Thiesen [8] and a mode system described by Skoglund and Wrigstad [11]. Both papers introduce the essential idea of qualifying references and method receivers with a `readonly` (or `read`) restriction. However, their discussions of the rules for `readonly` in the presence of arrays and generics do not include enough detail to be of any use to programmers, let alone language implementers.

Only the Javari language can be considered a complete solution to the problem of extending Java with a `readonly` qualifier because its specification (given first in [2] and improved in [12]) thoroughly discusses arrays, generics, and other features of Java that are affected by the addition of `readonly`. A helpful summary of the motivations for these three languages and the issues affecting them is given by Boyland [3]. He considers Javari as defined in [12] to be the best of the three because it is the only one to offer separate `assignable` and `mutable` qualifiers, but he suggests further improvements.

Javari's `@ReadOnly` qualifier takes the form of a Java 5 annotation and can be applied to any use of a reference type in a program, including method receivers. The opposite, `@Mutable`, is the default in most cases but can also be written explicitly. Here is a brief example of some legal and illegal statements in Javari, provided `StringBuffer` is suitably defined:

```
@ReadOnly StringBuffer sb = /* get from somewhere */;
int i = sb.length(); // OK, length can be called on a readonly reference
sb.append("foo");    // Illegal to call writing method on readonly reference

StringBuffer sb2 = sb;           // Illegal: attempts to remove restriction

StringBuffer sb3 = new StringBuffer();
sb3.append("bar");               // OK
@ReadOnly StringBuffer sb4 = sb3; // OK, adds restriction
```

By default, if an object is accessed through a `@ReadOnly` reference, its fields cannot be assigned, and those that have reference types behave as if qualified `@ReadOnly`; thus "`@ReadOnly`"-ness extends to sub-objects. The `@Assignable` qualifier is provided to exempt individual fields from the first restriction; to exempt a field from the second restriction, one may explicitly qualify it `@Mutable`.

One may express relationships between the mutabilities of the parameters, receiver, and return type of a method using the `@RoMaybe` qualifier. A method defined with `@RoMaybe` may be invoked with either of two signatures, one in which all occurrences of `@RoMaybe` are replaced by `@ReadOnly` and another in which they are replaced by `@Mutable`. Finally, `@QuestionRo` may be used in a type argument to represent a mutability wildcard. More detail about these qualifiers and the Javari type system may be found in [12].

Many experimental programming languages offer significant benefits to the user over better-known languages but have not come into widespread use because their designers placed too little emphasis on practicality. In contrast, as evidenced by the title of the first Javari paper [2], Javari was designed from the beginning as a practical language. A high priority has been placed on speeding its adoption by making it easy for Java programmers to learn and use Javari and by making its features available in some form in a future version of Java.

The original Javari implementation consisted of a modified version of the Java compiler that recognized an additional `readonly` keyword. However, Sun and others would consider it imprudent to add a keyword to Java or to support an additional type qualifier in the compiler itself because these changes could introduce bugs and incompatibilities into a language on whose stability users worldwide rely. Fortunately, the annotations introduced in Java 5 provide an excellent alternative. The Javari type checker will take the form of a compiler plugin that recognizes `@ReadOnly` annotations through the annotation processing API [5]. In other words, Javari code is merely Java code with extra annotations. The Java compiler itself ascribes no meaning to the annotations, but the Javari plugin uses them to check the code for type safety. Constructs in the code that are unsound according to the Javari type system generate "Javari unchecked warnings" similar to existing unchecked warnings for generics. Javari code will compile and run unchanged on an ordinary Java compiler, but the benefit of the extra type checking is lost.

A new Javari type checker following this model is planned. The original Javari compiler was based on Java 1.4; the generics introduced in Java 5 will make the new type checker considerably more complex. Furthermore, the annotations that Java currently provides are not powerful enough to serve as type qualifiers for a tool such as the Javari plugin, so extensions to the annotations facility were proposed [9]; a modified Java compiler that recognizes extended annotations is being developed. Meanwhile, the Javarifier, which automates the process of porting Java code to Javari by adding the appropriate annotations, is in development and has recently begun to work. A `@NonNull` qualifier (which guarantees that a reference is not `null`) is being implemented and studied alongside Javari because the two efforts involve mostly the same issues.

This project comes at an important time as the new Javari toolset is taking shape. It will examine the strengths and weaknesses of the toolset from the programmer's point of view and to determine how programmers can best take advantage of its features. The project will address the following questions:

- How much effort does it take to port a Java program to Javari? How can the Javarifier be used to speed the process?
- When one is converting a Java program to Javari by hand, in what cases is the best way to annotate a construct unclear? How should each such case be handled?
- How accurate are the annotations inferred by the Javarifier? What constructs, if any, lead it to infer incorrect annotations?
- How could the Javari tools be changed to make them easier for an average Java programmer to learn and use?
- How can the Javari tools be integrated with other software development tools, such as build systems and IDEs?

2 Methodology

To address the first four questions, we performed a case study by porting several existing Java programs to Javari. We began by manually annotating the "annotation scene library," which is part of the new Javari implementation; it can represent in memory the annotations on a set of classes and can read and write annotation index files [7]. We also needed to hand-annotate parts of the Java API so that the Javarifier could take them as given when analyzing other programs; otherwise it would have had to analyze the entire Java API first. We took notes on difficult cases in which it was not immediately clear how to correctly annotate the program, how these cases were resolved, and aspects of the Javari tools that made them difficult to use and therefore should be improved.

We originally intended to run the Javarifier on the annotation scene library to work out bugs in its implementation. This was to be in preparation for a comprehensive case study in which we would annotate one or more larger Java programs both by hand and using the Javarifier and then compare the results. However, we found the implementation of the Javarifier to be incomplete in ways that precluded applying it to large Java programs that make use of complex language features.

Thus, we decided instead to perform a case study on a smaller program in order to learn as much as possible about the Javari tools under the circumstances. Sălcianu and Rinard tested a purity analysis for Java [10] on the JOlden programs, which were originally developed as performance benchmarks for a prefetching technique [4]; since the purpose of the purity analysis is similar to that of Javari, we selected one of these programs, namely BH, for this case study.

We annotated BH first using the Javarifier and then by hand, noting the exact amount of time required for manual annotation. Each time the Javarifier's results and the manual results disagreed on the annotation for a program element, we reexamined the BH source code to determine which annotation was correct. By reconciling all of the differences in this manner, we created a set of “ideal” annotations for BH. (We believe that the ideal annotations are completely correct, but of course it is conceivable that we could have made a mistake during the reexamination.) Then we separately examined cases in which the Javarifier erred from the ideal annotations and cases in which the manual annotations erred from the ideal ones, and we classified the reason for each error.

We addressed the fifth question in the course of our other work by documenting the setup that we find most convenient for writing, building, and troubleshooting our own Javari programs. Since many Java programmers prefer to work in the Eclipse IDE, we placed particular emphasis on finding a convenient way to develop Javari programs in Eclipse.

3 Results and Discussion

3.1 Usability issues

We identified the following usability issues with the Javari and nonnull-checking tools:

Qualifier input. `@ReadOnly` and `@NonNull` qualifiers are long and inconvenient to type. We are considering abbreviating them to `@RO` and `@NN` respectively. In the meantime, many editors and IDEs support some form of abbreviation expansion; for example, Eclipse can be configured so that typing `ro` and pressing Ctrl+Space results in a `@ReadOnly` qualifier.

Bloated types. `@ReadOnly` and `@NonNull` make code less readable by bloating types, sometimes so much that they span more than one line. For example, the original annotation scene library used the type

`VivifyingMap<String, AClass<A>>` ;

when the library was annotated, the type became

`@NonNull VivifyingMap<@NonNull String, @NonNull AClass<A>>` .

Bloated types make the surrounding code more difficult to see. In some cases, one may be able to define a dummy subclass of a bloated type and use the subclass in its place as a typedef. Abbreviating the qualifiers would also mitigate the problem.

Default nullability. In the current implementation of the nonnull type checker, all references are nullable by default for the sake of backward compatibility. However, in our experience, the overwhelming majority of references should be nonnull, and if a reference is not marked `@NonNull`, it is more likely that the programmer made a mistake than that it really should be nullable.

It would be much more helpful to programmers if references were nonnull by default and each nullable reference had to be explicitly marked `@Nullable`, reminding the programmer to find out what the `null` value would signify and handle it appropriately. For example, the method

```
AnnotationBuilder AnnotationFactory.beginAnnotation(String annoTypeName)
```

of the annotation scene library returns `null` if the factory is uninterested in building annotations of the given type name. Annotating the return type `@Nullable` would decrease the probability that the programmer would miss the statement of this fact in the method's Javadoc comment. The nonnull type checker could consider references to be nonnull by default only in compilation units marked `@NonNullDefault`; that way, unannotated legacy code with all nullable references would compile unchanged.

Similarly, we found that most references should be `@ReadOnly`, but the fraction of references that should allow writing is significant enough that we do not propose the use of `@ReadOnlyDefault`.

Marking possible annotation sites. Much of the work of manually annotating a program consists of determining where in the source code annotations would be permitted; then it is usually easy to decide which annotation belongs in each slot. A tool that parses a source file and inserts a marker character in each place where an annotation would be allowed would be very helpful. In Eclipse, it would be more appropriate for the editor to display a graphical marker and show a pop-up menu of possible annotations when the user clicks the marker.

Casts. The target type of a cast does not automatically take on the type qualifiers of the value undergoing the cast. Therefore, the programmer must explicitly repeat the qualifiers, which is distracting because the purpose of the cast is to change the base type, not the qualifiers. For example, if `a` is a `@ReadOnly Object`,

```
@ReadOnly StringBuffer b = (StringBuffer) a
```

is illegal; the programmer must write

```
@ReadOnly StringBuffer b = (@ReadOnly StringBuffer) a .
```

It would be better if qualifiers on the type of the value undergoing the cast (here `@ReadOnly Object`) were preserved unless explicitly overridden by qualifiers on the target type of the cast. It would be nice to support casts containing only qualifiers, such as

```
@NonNull @ReadOnly Object b = (@NonNull) a ,
```

but we rejected this idea because it would require a change to Java syntax beyond merely allowing annotations in additional places.

3.2 Unclear cases during annotation

Many of the more complex areas of the software we studied, especially the Java standard library, presented problems when we tried to annotate them with `@ReadOnly` and `@NonNull`:

Mutability and type parameters. In the current definition of the Javari language, Java 5 type parameters carry their own mutability, meaning that if `T` is a type parameter, it is illegal to write `@ReadOnly T` or `@Mutable T`. While this design decision simplifies the language, it makes certain constructs difficult to annotate correctly.

The most compelling example is the type parameter of `java.lang.Class`. Should `StringBuffer.class` be a `Class<@ReadOnly StringBuffer>` or a `Class<@Mutable StringBuffer>`? Either way, it is impossible to write a generally useful signature for

`Class<T>.cast`. One would like to write

```
@RoMaybe T cast(@RoMaybe Object o) ,
```

but `@RoMaybe T` is illegal.

Javari could offer both type parameters that carry mutability and type parameters that do not. The same effect could be achieved perhaps more simply by offering separate mutability parameters and mutability-less type parameters; then the programmer could emulate a mutability-carrying type parameter by combining one parameter of each kind.

Unmodifiable implementations of mutable interfaces. `java.util.Collections.SingletonSet` is one example of an unmodifiable implementation of an interface containing mutating methods (namely `Set`). Should `SingletonSet` be marked `@Unmodifiable`, forcing all references to `SingletonSet`s to be `@ReadOnly`? (Javari currently does not allow an `@Unmodifiable` class to have a non-`@Unmodifiable` supertype, but the rule could be changed.)

If so, from the perspective of Javari, there is no need for `SingletonSet` to implement the mutating methods of `Set` because it is impossible to obtain an `@Mutable SingletonSet` on which these methods could be called. It would be most convenient if the `SingletonSet` source code did not have to mention these methods at all. However, Java itself requires `SingletonSet` to provide these methods, just like any other class that implements `Set`. It would be nice if stub mutating methods that throw exceptions could be generated automatically, but this is outside the scope of what the Javari type-checker, as a compiler plugin, can do. Thus, the programmer must write the stub methods explicitly, and the Javari type-checker needs a special exception to permit such stub methods to have mutable receivers even in an unmodifiable class.

`Collections.UnmodifiableSet` is the canonical example of an unmodifiable implementation of a mutable interface. It might seem that `UnmodifiableSet` has no place in Javari since "wrapping" a set in a statically checked `@ReadOnly` qualifier is superior to wrapping it in a proxy object. However, runtime checks are needed to protect an object from other code in the JVM that might perform operations that are unsound according to Javari. For example, runtime checks are needed to prevent untrusted Java applets from compromising privileged parts of the Java standard library by circumventing their `@ReadOnly` qualifiers. It would be helpful to programmers writing security-critical code if the Javari plugin could verify that a `@ReadOnly` reference is never passed out to untrusted code without first being wrapped in a proxy to protect the target object from mutation. Perhaps Javari could allow a reference to be qualified `@CheckedReadOnly`, meaning that a runtime check prevents one from doing any harm by forcibly calling mutating methods of the target.

Generics and read-only restrictions are very similar in some respects. In fact, Java offers a `CheckedSet` class analogous to `UnmodifiableSet`, and a `@CheckedGeneric` qualifier analogous to `@CheckedReadOnly` might be useful.

Iterators. What does it mean for an iterator to be mutable—that one can advance it with `next()` or that one can remove elements from the underlying collection with `remove()`? We decided that, since an iterator that cannot be advanced is essentially useless, `next()` should be allowed even on read-only iterators and the mutability of an iterator should follow that of the underlying collection.

However, if iterators had separate `advance()` and `retrieve()` methods or could be cloned, iterators that cannot be advanced would not be completely useless, and this decision would have to be reconsidered. If Javari supported mutability parameters (see above), an iterator's mutability could govern whether it may be advanced, and a separate mutability parameter could be used for the mutability of the underlying collection.

Dynamic discovery of arrays. The `deepEquals`, `deepHashCode`, and `deepToString` methods of `java.util.Arrays` are examples of methods that need the ability to determine at runtime whether an object is an array and, if so, read its elements. If such a method is interrogating a `@ReadOnly Object o` and finds that `o instanceof Object[]`, `o` could have originally been created as either a `@ReadOnly Object[]` or a `@Mutable Object[]`. Thus, since Javari treats an array element type as a special case of a generic type argument, the method should cast `o` to a `@QuestionRo Object[@ReadOnly]`. This will allow it to read out individual elements of `o` as `@ReadOnly Objects`, and then it may recurse on those elements.

3.3 The BH case study

The BH JOlden program consists of a total of seven classes and 1130 lines of code implementing the Barnes-Hut force calculation algorithm [1]. First we ran the Javarifier on the code. Then, we manually annotated the code in Eclipse using abbreviation expansion as described in Section 3.1. Furthermore, we used commented-out annotations (`/*@ReadOnly*/`) that do not cause a syntax error in Eclipse but are still recognized by the extended compiler.

It took 30 minutes to manually annotate BH, which was considerably longer than we expected. Since we had no prior knowledge of the code and the documentation provided little help in determining which references were meant to be read-only and mutable, we based our manual annotation of each reference on the places in which that reference was used, and we found ourselves relying heavily on Eclipse's Mark Occurrences and Search for References features. In this setting, it appears that a human cannot possibly have an advantage over the Javarifier, which uses essentially the same techniques that we used but operates automatically.

Where the Javarifier results and the manual results differed, we inspected the BH source code again to determine the correct annotations. This process produced the ideal annotation set to which we compared both the Javarifier results and the manual results. Table 1 shows the number of occurrences of each mutability annotation in the ideal annotation set on or inside the types of BH program elements of each kind. In the 1130 lines of code, there were 205 types or type layers that could carry mutabilities, or about one for every five lines.

Errors in manual annotations. We found that 22 (11%) of the manual annotations

	ReadOnly	Mutable	ThisMutable	RoMaybe	QuestionRo	Total
Field	2	0	19	2	0	23
Receiver	18	30	0	6	0	54
Parameter	32	13	0	0	0	45
Return type	4	13	0	12	0	29
Local variable	4	48	0	2	0	54
Total	60	104	19	22	0	205

Table 1: Occurrences of mutability annotations on BH program elements in the ideal annotation set.

were incorrect (i.e., they differed from the ideal annotations). In five of these cases, we forgot to consider the receiver of an obviously read-only method (such as `toString()`). In most of the remaining cases, we had not investigated the logic of BH thoroughly enough to know how the element in question was intended to be used.

Errors in Javarifier annotations. 35 (17%) of the Javarifier's annotations were incorrect. A bug in the implementation of the Javarifier, namely that it does not constrain mutabilities of type arguments of `extends` and `implements` clauses correctly, led to 9 of these errors.

In four cases comprising a total of 12 annotations, we used `@RoMaybe` to make a method's signature more general. However, since the rest of the code did not actually make use of the general signature, the Javarifier incorrectly inferred a more specific signature. Similarly, during manual annotation, we placed 9 `@ThisMutable` and `@Mutable` annotations on fields and local variables to document that mutating them would do no harm, even though the program did not actually mutate them; the Javarifier annotated them `@ReadOnly`. Since the Javarifier annotates references solely based on how they are used by the rest of the program, that it cannot detect the programmer's intent in these cases is an inevitable limitation.

A construct that Javari could not handle. BH threads `Body` objects into a doubly-linked list using previous and next pointers embedded in the objects, and the method `Body.elements()` returns a `java.util.Enumeration` that traverses the list. We manually annotated the previous and next fields this-mutable, wrote an appropriate signature for `elements()` using `@RoMaybe`, and attempted to annotate the inner class used to implement the enumeration. However, when we saw the Javarifier's results, we realized that the manual annotations were erroneous; in fact, we could not find a correct way to express the inner class in Javari without significant rearrangement to the program. This issue affected 10 annotations, which we did not count as errors in either the manual annotations or the Javarifier's annotations.

The BH case study should be seen as a lesson about when it is *not* appropriate to convert a Java program to Javari. We believe mutability annotations to be an excellent addition to the internal interfaces of a large application because they are useful as documentation to the programmers and they ensure that restrictions placed on mutation of objects by one part of the program are obeyed by other parts of the program. In contrast, BH is a relatively small simulation engine without anything that could be described as a clear internal interface, except perhaps for the public interface of class `MathVector`. Thus, mutability annotations in BH are of little value, whether they are added manually or automatically. Furthermore, it is inappropriate to undertake the annotation of a program of which one has no significant prior knowledge. If the program is needed as a library, we recommend creating stubs by annotating just the signatures of the public members of the library; the Javarifier can make use of such stubs when analyzing clients of the library.

3.4 Our Javari development environment

We performed all Javari development and manual annotation using Eclipse's Java editor. The annotation scene library was initially written as an Eclipse Java project built using the Eclipse Java builder. Commented-out Javari annotations could be added easily without interfering with building and running the library in Eclipse. When we desired class files of the library containing the Javari annotations, we built the library in a separate output folder using the extended compiler.

The setup for the annotated Java API was similar. We used an Eclipse Java project built in a dummy output folder to take advantage of Eclipse's automatic marking of syntax errors and its helpful navigation features. We then manually invoked the extended compiler to rebuild the annotated API in a separate output folder each time we made a change.

It is very easy to invoke the extended compiler from Ant: one need only specify the path to its executable, which we named `xjavac`, in the `<javac>` task. Thus, we wrote an Ant build file for each project that contained extended annotations. Eclipse supports Ant as an external tool builder, so Eclipse could be configured to run the extended compiler in parallel with its own Java builder every time a project is changed. In fact, since version 3.2, Eclipse can load the errors and warnings produced by a Java compiler during an Ant build into the Problems view. While we preferred to invoke Ant from the command line, end users might wish to take advantage of these features to integrate the extended compiler with Eclipse.

Some of the Javari tools have many dependencies (for example, the Javarifier requires the annotation scene library and modified versions of the ASM and Soot libraries), and we eventually tired of manually maintaining a compile-time classpath in each Ant build file and a runtime classpath in the launcher script for each Javari tool. Thus, we created `pathtool`, an auxiliary script that automatically derives these classpaths from Eclipse `.classpath` files. While `pathtool` is equally applicable to Java and Javari programs, it is more likely to be necessary for Javari programs because they cannot be built entirely within Eclipse.

4 Conclusion

We have presented a thorough study of the practical issues surrounding the Javari programming language and toolset, based on experience gained by porting several Java programs to Javari. Javari is an extension of Java with a `@ReadOnly` type qualifier and related safety checking that prevents certain kinds of bugs. We have discussed how to express existing Java code in Javari in certain difficult cases, ways in which the usability of the tools could be improved, and the outcome of a case study comparing two techniques used to convert BH, a Java program, to Javari. Finally, we have given suggestions for setting up a Javari development environment. We trust that the improvements to Javari that we propose and the practical advice that we present will speed Javari's adoption by today's programmers and thereby bring the benefits of more reliable software to the public.

5 Acknowledgments

I would like to thank my mentor, Prof. Michael Ernst of MIT's Computer Science and Artificial Intelligence Lab, for arranging my participation in the Javari project and for his helpful comments at all stages of my work. I would like to thank Matt Papi, Jaime Quinonez, and Matthew Tschantz of the Javari team for helping me carry out the study and for developing the tools it required. I would like to Marc Horowitz, Scott Kominers, Jenny Sendova, Shrenik Shah, and Quentin Smith for their help in the preparation of my paper. I would like to thank the Center for Excellence in Education for making this research opportunity possible. Finally, I would like to thank my parents for their ongoing advice and support.

References

[1] J. Barnes and P. Hut. A hierarchical $O(N \log N)$ force-calculation algorithm. In: Nature 324 (1986), no. 4, 446–449.

[2] A. Birka and M. D. Ernst. A practical type system and language for reference immutability. In: *Proceedings of Object-Oriented Programming, Systems, Languages and Applications 2004.* 35–49.

[3] J. Boyland. Why we should not add `readonly` to Java (yet). In: Journal of Object Technology 5 (2006), 5–29.

[4] B. Cahoon and K. S. McKinley. Data flow analysis for software prefetching linked data structures in Java. In: *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques.* Barcelona, Spain (2001).

[5] J. Darcy. JSR 269 (Pluggable Annotation Processing API) Public Review Draft v0.69. Available at `http://jcp.org/aboutJava/communityprocess/pr/jsr269/index.html` (2006/07/09).

[6] J. Gosling, B. Joy, G. L. Steele, and G. Bracha. *The Java Language Specification, Third Edition.* Available at `http://java.sun.com/docs/books/jls/third_edition/html/j3TOC.html` (2006/06/28).

[7] Javari team. Annotation index file specification. Unpublished typescript (2006). Available at `http://pag.csail.mit.edu/javari/index-file-format.pdf` (2006/07/28).

[8] G. Kniesel and D. Theisen. JAC—access right based encapsulation for Java. In: Software—Practice and Experience 31 (2001), no. 6, 555–576.

[9] M. M. Papi and M. D. Ernst. Annotations on Java types. Unpublished typescript (2006). Available at `http://pag.csail.mit.edu/javari/java-annotation-design.pdf` (2006/07/28).

[10] A. Sălcianu and M. Rinard. Purity and side effect analysis for Java programs. In: *Proceedings of the 6th International Conference on Verification, Model Checking and Abstract Interpretation.* Paris, France (2005).

[11] M. Skoglund and T. Wrigstad. A mode system for read-only references in Java. In: *Proceedings of the 3rd Workshop on Formal Techniques for Java Programs.* (2001).

[12] M. S. Tschantz and M. D. Ernst. Javari: adding reference immutability to Java. In: *Proceedings of Object-Oriented Programming, Systems, Languages and Applications 2005.* 211–230.