Pluggable Type-checking for Custom Type Qualifiers in Java

Matthew M. Papi Mahmood Ali Telmo Luis Correa Jr. Jeff H. Perkins Michael D. Ernst
MIT Computer Science and Artificial Intelligence Lab, Cambridge, MA, USA
{mpapi,mali,telmo,jhp,mernst}@csail.mit.edu

ABSTRACT

We have created a framework for adding custom type qualifiers to the Java language in a backward-compatible way. The type system designer defines the qualifiers and creates a compiler plug-in that enforces their semantics. Programmers can write the type qualifiers in their programs and be informed of errors or assured that the program is free of those errors. The system builds on existing Java tools and APIs.

In order to evaluate our framework, we have written four type-checkers using the framework: for a non-null type system that can detect and prevent null pointer errors; for an interned type system that can detect and prevent equality-checking errors; for a reference immutability type system, Javari, that can detect and prevent mutation errors; and for a reference and object immutability type system, IGJ, that can detect and prevent even more mutation errors. We have conducted case studies using each checker to find real errors in existing software. These case studies demonstrate that the checkers and the framework are practical and useful.

Categories and Subject Descriptors

D3.3 [**Programming Languages**]: Language Constructs and Features—*data types and structures*; F3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs; D1.5 [**Programming Techniques**]: Object-oriented Programming

General Terms

Languages, Theory

Keywords

annotation, compiler, Java, javac, NonNull, type qualifier, type system, verification

1. Introduction

Types help to detect and prevent errors: types help programmers to organize and document data, and types allow tools to verify that a program does not violate the type system's constraints. However there is often much information about a type that a programmer cannot express. User-defined type qualifiers can enrich the built-in type system, permitting more expressive compile-time checking and guaranteeing the absence of certain errors.

Type qualifiers are modifiers that provide extra information about a type or variable. We present a framework that allows a type system designer to define new type qualifiers, and to create compiler plug-ins that check their semantics and issue lint-like warnings. A programmer can then use these type qualifiers throughout a program to obtain additional guarantees about the program. The type system defined by the type qualifiers does not change Java semantics, nor is it used by the Java compiler or run-time system. Rather, it is used by the checking tool (a compiler plug-in), which can be viewed as performing type-checking using this richer type system. (The qualified type is usually treated as a subtype or a supertype of the unqualified type, but our system is general enough to accommodate type systems that do not abide by this restriction or that add additional rules.) Since they are user-defined, developers can create and use the type qualifiers that are most appropriate for their software.

```
1  @NonNullDefault
2  class DAG {
3    Set<Edge> edges;
4
5    // ...
6
7    List<Vertex>
8    getNeighbors(@Interned @ReadOnly Vertex v) @ReadOnly {
9      List<Vertex> neighbors = new LinkedList<Vertex>();
10     for (Edge e : edges)
11       if (e.from() == v)
12         neighbors.add(e.to());
13     return neighbors;
14   }
15 }
```

Figure 1: Part of a `DAG` class that represents a directed acyclic graph. The code illustrates how a programmer may use three type qualifiers: `@NonNull`, `@Interned`, and `@ReadOnly`. Then, a type-checking plug-in detects, or verifies the absence of, program errors.

Figure 1 shows example uses of three type qualifiers. **(1)** The `@NonNullDefault` annotation (line 1) indicates that no reference in the `DAG` class may be null unless otherwise annotated. It is equivalent to writing line 3 as "`@NonNull Set<@NonNull Edge> edges;`", for example. This guarantees that the uses of `edges` (line 10) and `e` (lines 11 and 12) cannot cause a null pointer exception. Similarly, the `@NonNull` return type of `getNeighbors` (line 7) enables its clients to depend on the fact that it always returns a `List`, even if `v` has no neighbors.
(2) The two `@ReadOnly` annotations on method `getNeighbors` (line 8) guarantee to clients that the method does not modify its `Vertex` argument or its `DAG` receiver. The lack of a `@ReadOnly` annotation on the return value (line 7) indicates

that clients are free to modify the returned `List`.
(3) The `@Interned` annotation on line 8 (along with an `@Interned` annotation on the return type in the declaration of `Edge.from`, not shown) indicates that the use of object equality (`==`) on line 11 is a valid optimization. In the absence of such annotations, use of the `equals` method is preferred to `==`.

Our key goal is to create a type qualifier framework that is compatible with the Java language, virtual machine (JVM), and toolchain. Previous proposals for Java type qualifiers are incompatible with the existing Java language or tools, are too inexpressive, or both. We found that integrating with the Java toolchain required extra engineering effort, but resulted in better usability, robustness, maintainability, and portability. Our framework builds upon 4 key Java technologies: the JSR 175/308 annotations syntax, the JSR 269 annotation processing API, the Tree API for compiler ASTs, and the JSR 199/269 error reporting mechanism. No modifications to the virtual machine are necessary.

Our framework uses Java annotations to express type qualifiers. Java 5 annotations [4] are too inexpressive, so we use the JSR 308 [14] annotation extension, which is planned for inclusion in Java 7. JSR 308 permits annotations to be written on any use of a type, including generic type arguments, array elements, method receivers, class literals, casts, etc. JSR 308 specifies the syntax of the annotations, but not their semantics (that is the role of our framework). JSR 308 also extends the Java class file format so that all annotations are represented in the class file. This permits type-checking against binary versions of annotated classes (e.g., an annotated library) and will facilitate type-checking of Java bytecode. The JSR 308 changes are backward-compatible with Java 5. Furthermore, the JSR 308 compiler permits annotations to be enclosed in comments (`/*...*/`) so that Java code that uses these annotations can be processed by compilers for earlier Java versions. Our framework requires no compiler changes beyond these planned for inclusion in Java 7. Our framework is currently implemented for Sun's javac, which is the standard Java compiler. Eclipse and other IDEs can use javac as their compiler.

In our framework, a type-checker is implemented as a compiler plug-in, using the standard annotation processing API [8]. A compiler plug-in (also known as an "annotation processor") is invoked by specifying one extra compiler argument (see Section 2). Use of this API means that programmers do not need to use an external tool (or, worse, a custom compiler) to obtain the benefits of type-checking; running the compiler and fixing the errors that it reports is part of ordinary developer practice. Use of the annotation processing framework rather than hard-coding changes deep in the compiler implementation required re-implementing some compiler functionality in our framework, but the result is more modular, is easier to extend to new type qualifiers, and is less sensitive to javac changes.

The Tree API (part of Sun's Mustang Java 6 implementation) gives a compiler plug-in access to the program's AST. The plug-in uses this to visit each expression, method declaration, and other elements of the program, performing type-checking as it goes.

A compiler plug-in reports warnings and errors through the same reporting mechanism that the compiler itself uses [35, 8]. As a result, all errors are displayed in a uniform fashion and the compiler is aware of whether a warning or error has been issued during a particular compilation.

To demonstrate the practicality of our system, we have developed type-checkers for four useful type qualifiers and have performed case studies with each checker. The NonNull qualifier provides an implicit subtype of each Java type that excludes the value null (i.e., a reference whose type is NonNull can never be null, and dereferencing it can never throw a null pointer exception); see Section 3. The Interned qualifier provides an implicit subtype denoting that a variable contains a canonical value and may be safely tested using the `==` operator (versus the `equals` method); see Section 4. The Javari [33] language provides a ReadOnly type qualifier for reference immutability (a reference may not be used to modify its referent) that is an implicit supertype of each Java type; see Section 5. The IGJ [37] language provides ReadOnly and Immutable type qualifiers that permit specification of both reference immutability and object immutability (an object may not be mutated through any reference to it) and that includes qualified types that are neither supertypes nor subtypes of the unqualified Java types; see Section 6.

Although some of the four type systems described above are conceptually simple, building a scalable version of each one required flexibility in our framework: the NonNull checker implements flow-sensitivity; the Interned checker treats enumerated types and primitives specially; and the reference immutability checkers implement type systems that are more than a simple sub- or super-type of existing Java types.

The four type-checker plug-ins (non-null, interned, Javari, and IGJ), the checkers framework, and the JSR 308 extended annotations Java compiler are publicly available for download from the JSR 308 web site, `http://pag.csail.mit.edu/jsr308`. We anticipate that our tools will both aid practitioners and also permit researchers to experiment with new type systems in the context of a realistic language, Java.

2. Implementation

Java 6 includes an API for pluggable annotation processing [8], which is invoked via the `-processor` option to javac. Our framework extends this functionality to permit annotation processors (which we also call "type-checking compiler plug-ins" or just "checkers") to be run after the type resolution/attribution compiler phase. It is used as follows:

```
javac -typeprocessor NonnullChecker MyFile.java
```

Successful type-checking may require annotation of the signatures of external libraries that are called by the code being checked, and we have a suite of tools for manipulating such annotations. Use of `@SuppressWarnings` annotations and command-line arguments permits suppressing warnings by statement, method, class, or package. Naturally, suppression of warnings compromises the checker's guarantee that the analyzed code is error-free. Additionally, the framework does not reason about the target of reflective calls.

Our checkers framework extends Java's annotation processing API [8] for compile-time type-checking of type qualifier annotations. The framework provides several features that reduce the time and effort required to create a new type-checker. First, it provides data structures for querying the annotations on a program element regardless of whether that element is found in a source file or in a class file. Second, it provides a template (using the visitor design pattern) for

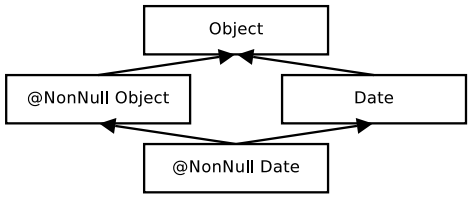

Figure 2: Type hierarchy for the NonNull type qualifier.

applying a type qualifier's rules to an input program, and it interfaces this component to the Java compiler. Third, the framework uses the Java compiler's messaging interface for reporting and collecting errors during type checking. Finally, the framework provides additional utilities for qualifiers that are either subtypes or supertypes of the unqualified type. For instance, NonNull and Interned types are subtypes of their unqualified types, and ReadOnly types are supertypes of their unqualified types.

A checker consists of three classes. The *compiler interface* class provides hooks that are called by the annotation processing facility [8], and it reports errors via the compiler's messaging mechanism [35]. The *visitor* class is a visitor for Java source syntax trees as provided by the Tree API. The visitor class performs type-checking as it walks each source file's AST. The *type factory* is used by the visitor to read annotated types out of the AST.

The checkers framework provides an abstract visitor class that performs checking based on the type hierarchy induced by the type qualifier. For example, it is illegal to assign a supertype to a subtype, so both of these lines are illegal (assuming the obvious declarations):

```
myInternedObject = myObject;  // invalid assignment
myObject = myReadOnlyObject;  // invalid assignment
```

The abstract visitor also checks method arguments, receivers, return values, and overriding.

Use of the checkers framework greatly simplifies writing a compiler plug-in. The visitor class for the NonNull checker (Section 3) overrides only one method, to warn about dereferences of possibly-null expressions. The NonNull checker's type factory contains a flow-sensitive analysis that performs NonNull inference after null checks, dereferences, etc. The visitor class for the interned checker (Section 4) overrides three methods: one that warns if either argument to `==` or `!=` is not interned, and two that implement inference of @Interned annotations for final variables that are initialized to an interned value. The interned checker's type factory marks all primitives, `enum` types, String literals, etc. as interned.

3. Nullness checker for null pointer errors

The NonNull checker implements a simple qualified type system in which, for every Java class `C`, `@NonNull C` is a subtype of `C` (see Figure 2). As an example of the difference, A reference of type `Boolean` always has one of the values `TRUE`, `FALSE`, or `null`. By contrast, a reference of type `@NonNull Boolean` always has one of the values `TRUE` or `FALSE` — never `null`. Dereferencing an expression of type `@NonNull Boolean` can never cause a null pointer exception.

The NonNull checker issues a warning in two cases:

1. When an expression of non-NonNull (i.e., Nullable) type is dereferenced, because it might cause a null pointer exception.

Program	(Default)	Size Files	Lines	ALocs	Anno-tations	Warnings errs	AI	TW
Anno. file utils	(Nul)	49	4640	3700	699	3	23	12
Lookup	(Nul)	8	3961	1757	248	7	7	16
Lookup	(NN)	8	3961	1757	83	7	7	11
NonNull checker	(NN)	7	1031	406	65	5	3	0
Chk. framework	(NN)	21	5451	2513	308	29	50	20

Figure 3: Statistics from case studies of NonNull type-checker plug-in. Columns are explained in Section 3.

2. When an expression of NonNull type might become null, because it is a misuse of the type.

The following code example illustrates both kinds of errors.

```
         Object   obj;  // might be null
@NonNull Object nnobj;  // never null
...
obj.toString();    // possible null pointer exception
nnobj = obj;       // nnobj may become null
```

The NonNull checker performs intraprocedural flow-sensitive inference to determine which expressions of Nullable type can be statically determined to be NonNull. In other words, it determines when the declared type is wider than (is a supertype of) the actual type. This enables programmers to omit some annotations; it enables a single variable to have different qualified types in different parts of its scope; and in both cases, it suppresses false warnings that the checker would otherwise emit.

For each Nullable reference used in a method, the flow-sensitive analysis propagates a bit throughout the AST to indicate, at each statement, whether that reference is known to be NonNull. The analysis is straightforward. A reference is known to be null after an explicit null check in an assert statement or a conditional, or after a NonNull value is assigned to it. Such a reference is known to remain null until it is reassigned (including assignments to possibly-aliased variables and calls to external methods) or until flow rejoins a branch where the variable is not known to be null. The flow-sensitive analysis greatly decreases the annotation burden, and can completely eliminate the need to annotate a method body.

3.1 NonNull case study

We evaluated the NonNull checker via case studies on 4 programs totaling 15083 lines of code (see Figure 3). The annotation file utilities extract annotations from, and insert them in, source and class files. Lookup is a featureful paragraph grep utility; we also annotated the bodies of library classes that it uses, except those in the JDK. The NonNull type-checker and the checkers framework are the subject of the current paper. We also annotated the signatures of 48 library classes that the programs use (mostly from the JDK).

The columns of Figure 3 give the default used by the checker (Nullable or NonNull); the program size (in files, lines, and potential annotation locations); the number of annotations we wrote; and the number of warnings issued by the checker (see below).

To obtain a broader variety of qualitative experience, we treated the programs in several ways. We annotated the annotation file utilities without the benefit of the NonNull checker, then checked the results of the hand annotation.

For all other programs, we incrementally annotated the program and fixed errors reported by the NonNull checker. We annotated some of the programs using `@Nullable` as the default, and others using `@NonNull` as the default; we annotated the Lookup program in both ways to better compare the two defaults.

Warnings issued by the NonNull checker fall into three categories (see Figure 3). User errors ("errs") are bugs — serious problems that may lead to null-pointer exceptions at run time. Application invariants ("AI") are uses of a possibly-null type in a context where the value cannot be null, due to an application-specific property that is inexpressible in the type system. We suppressed warnings of this type by adding a run-time assertion (e.g., `assert x!=null;`) for each application invariant. Tool weaknesses ("TW") also lead to false positive warnings; nearly every one of these resulted from lack of generic type inference for type qualifiers and wildcards. For example, Java infers the generic type `T` in the return type `Set<T>` of the `Collections.singleton()` method, but the NonNull checker does not yet infer whether `T` is NonNull. Users are always permitted to specify the generic type of a method, and doing so eliminates these warnings.

In the Annotation File Utilities case study, there was a fourth cause of checker warnings: user omissions in which the programmer forgot to write a `@NonNull` annotation for a NonNull type; there were 31 of them, all of which were easily fixed after running the checker. Even though the programmer had thought about the NonNull annotations (and had fixed a number of bugs while adding them), 3 bugs remained that the checker caught.

The most common cause of errors was failure to check that a value returned by a method was non-null. Most of the bugs in the NonNull checker could be reduced to the following form (which is the same form as most NullPointerException bug reports we have received from users of the checker):

```
if (x.y().z())
  somethingOptional();
```

where the return type of `x.y()` is Nullable, and `somethingOptional` need not be called if `x.y() == null`. In many of these cases, `y` was a method in an instance of a library class to which `x` referred, and `y` returned `null` for only a tiny percentage of valid inputs. A representative example is processing of a `static` block, as in:

```
class C {
      // ...
      static {
        // ...
      }
}
```

javac ultimately compiles the contents of the static block into a static initializer method. However, a library method for determining the enclosing method of a statement returns `null` (rather than data for the static initializer) when passed a statement in the static block. This is an unusual case that did not appear in our test suite, but it is important that the framework and the checkers not crash when processing that code. One other bug in the NonNull checker was introduced by a bug fix that added code using variable `x` before the null check for `x`; we fixed this by reordering the statements.

Here are some example errors from the Lookup program. The `deleteDir` utility method throws a null pointer exception if passed a filename that is not a directory, because `File.listFiles` returns null in that case. A `readLine` method can throw a null pointer exception because `Matcher.group` can return null; checking for null permits a comprehensible error message rather than a crash, and pointed out several similar (but not null-related) problems in the same option processing code. A final problem we discovered with Lookup was inability to remove long options, because null is used as a flag.

In each of the programs we annotated, we were able to remove dead code resulting from null checks of values that cannot be null.

We ran the FindBugs [25] null pointer checker on the Lookup program, in which our type-checker found 7 errors. FindBugs reported 1 error (a false positive) and 1 warning (dead code from a null check of a value that cannot be null). We also ran JLint (`http://jlint.sourceforge.net/`), which reported 8 null-related messages, all of them false positives.

We re-annotated the Lookup program using a default of `@NonNull` rather than `@Nullable`. (The two Lookup case studies revealed the same errors, as indicated in Figure 3.) We felt that the NonNull default made the code easier to read, but it was sometimes easy to overlook default `@NonNull` annotations, especially on generic types. Another positive effect of the NonNull default is that it default biases users away from using Nullable variables; such variables have a place in all programs, but should be avoided when possible.

Both our own experience and the literature [7] led us to believe that a NonNull default would reduce the number of annotations. This was true for fields and method calls, which made the code more readable; annotations point out exceptions rather than the normal case. However, we had to annotate *more* local variables. When the default was Nullable, the flow-sensitive analysis often inferred NonNull types for local variables (which are subtypes of the declared Nullable types), so an unannotated method body often type-checked. However, when the default was NonNull, it is not sensible for the checker to silently treat a variable as a supertype of its declared NonNull type, and we had to annotate every local variable that could hold null. One solution to this problem would be to use Nullable as the default for local variables, and NonNull for all other locations. A conceptually similar solution would be to perform type inference for all local variables; this could be intraprocedural, since we may assume that all called methods are fully annotated. A problem with both approaches is that an unannotated type such as `Integer` would mean two different things in different parts of a single method, which might be confusing. We plan to experiment with such an approach to see how users react to it.

Our biggest frustration when using the NonNull checker was the inability to express application invariants in the NonNull type system. Characteristic examples from the Lookup program are that `entry_stop_re` is null if and only if `entry_start_re` is null, and that a variable holding a factory method for a class is non-null if the class has no constructor and the class is not an enum. Our NonNull type system is good at stating whether a variable can be null, but not at stating the circumstances under which the variable can be null. Expressing these application invariants would require a substantially more sophisticated system, such as dependent types [31]. We used null checks to suppress each checker warning, which had the positive side effect of checking the property at run time. We were very happy with the errors

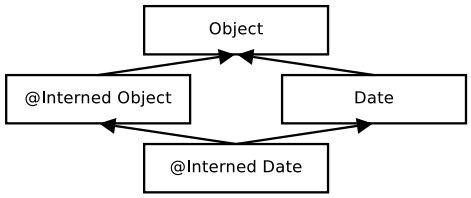

Figure 4: Type hierarchy for the Interned type qualifier.

that our system found, but use of the NonNull checker was addictive: we wanted to eliminate even more possibilities for error, such as the possibility of run-time assertion failures if we had mis-stated the conditions.

The compiler API used by the NonNull checker contains a number of methods that return null if and only if their single parameter is null. For these methods (and some JDK methods such as `Class.cast` and `Properties.getProperty`), a `@NNMaybe` qualifier similar to the `@ROMaybe` qualifier of Javari (Section 5) would have been useful.

4. Interned checker for equality-testing errors

Interning, also known as canonicalizing or hash-consing, finds or creates a unique concrete representation for a given abstract value. That representative can be used in place of any other concrete representation. For example, many `String`s could represent the 11-character sequence `"Hello world"`. Interning selects a particular one of these as the canonical one; a client should use that one in preference to all others.

Interning yields both space and time benefits. The space benefit stems from the fact that many references can point to the same representation. The time benefit stems from the ability to use `==` instead of `equals()` for comparisons. As another benefit, `x == y` is more readable than `x.equals(y)`, especially for complex expressions, and the equality test reminds the reader of the invariants on the underlying data structure. The `intern` operation has a modest time and space cost, so if few duplicate objects are created, or few comparisons are performed, interning may not be beneficial. Another potential problem is that failure to intern can lead to bugs: use of `==` on distinct objects representing the same abstract value may return false, as in `new Integer(22) == new Integer(22)` which yields `false`.

4.1 Interned checker

We have written a checker for an `@Interned` type annotation. The Interned type system is similar to that for NonNull; see Figure 4. If the plugin issues no warnings for a given program, then all reference equality (`==`) tests in that program operate on interned types. The checker issues a warning in two cases:

1. When a reference (in)equality operator (`==` or `!=`) compares objects and the type of at least one operand is not @Interned.
2. When a non-Interned type is used where an Interned type is expected.

The following code example illustrates both kinds of errors.

```
          String  s;
@Interned String is;
...
if (s == is) { ... } // warning: unsafe equality
is = s;              // warning: unsafe assignment
```

An `@Interned` annotation indicates that the annotated reference refers to an interned value, although other instances of the class may be uninterned. `String`s are often used in this way, since clients can know which strings may be used in multiple contexts. An annotation on the class declaration indicates that every instance of a class is interned, just as for Java enumerated classes defined with `enum`. String literals, primitives, enumerated classes, the null literal, the result of `String.intern`, and user-annotated object creation expressions (using `new`) are considered interned.

The interned checker does not require flow-sensitivity. It also requires no library annotations, since the only library method that affects interning is `String.intern`.

4.2 Interned case study

We evaluated the Interned checker by applying it to the source code of Daikon [13]. Daikon is a dynamic invariant detector — that is, it observes program executions and generalizes from observed values to likely invariants. Daikon consists of approximately 250KLOC of Java code.

Memory usage is the limiting factor in scaling Daikon to larger programs [30]. To conserve memory, the Daikon implementation uses the interning design pattern [22] extensively. 1170 lines of comments or code contain "canonical" or "intern" (or a variant of those words, but not counting unrelated words such as "internal"). Over 200 run-time assertions check that values are properly interned; 67 of those have no other purpose (e.g., `x==x.intern()`); 137 others can be viewed as checking both interning and other types of data consistency (e.g., `x.ppt==y.ppt`). Daikon contains an `intern` or `canonicalize` method for 10 classes, including both classes defined in Daikon and static interning methods for types defined elsewhere such as Integer and arrays. The Daikon developers use an Emacs plug-in that checks code for String-related interning errors whenever a file is saved.

We annotated 11 files (12KLOC) in Daikon with 127 `@Interned` annotations.[1] We introduced 3 new local variables to replace code of the form `x=x.intern()`. We also added 14 `@SuppressWarnings` annotations to eliminate false positive (and in a few cases also a local variable, to reduce the scope of the warning suppression): 5 due to lack of generic type inference in the type-checker; 4 due to calls to other files that had not yet been annotated; 3 due to casts in `intern` methods; 1 due to an application invariant (checking whether a variable was still set to its initial value); and 1 due to a checker bug.

This effort revealed 9 previously unknown interning-related errors in Daikon, 2 performance bugs (unnecessary interning), and a design flaw. We fixed all but the latter. We briefly describe these problems.

The `DeclReader.read_data` method, which reads trace files, returned interned data in 4 places and uninterned data in 2 places. However, a client (`WSMatch`) sometimes used `==` for comparisons of uninterned results. We added 2 missing calls to intern methods in `read_data`, so its result is always interned.

[1] Our case study focused on the parts of the code that make the most use of interning. 72% of the files have no interning comments/calls, and 87% have 0, 1, or 2. Thus, most files require no `@Interned` annotations. The files we annotated contain more than half of Daikon's 1170 interning comments/calls.

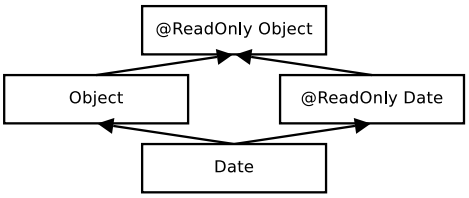

Figure 5: Type hierarchy for Javari's ReadOnly type qualifier.

A code comment indicated that the `VarInfo.str_name` field was interned, but there were 5 errors in `VarInfo` constructors. The uninterned field values escaped via the `name()` method (also commented as interned) to many clients that tested them with `==`.

The `VarInfo.var_info_name` field is also interned. Method `simplify_expression` performs algebraic simplification by side effect (which is necessary for preserving object equality). The method contains 17 branching points and fails to re-intern the new value of `var_info_name` in 2 locations.

In another case there is too much, not too little, interning. Method `FileIO.read_data_trace_record` is the inner loop of trace file reading. It interned lines as they were read from a file, but this interning was taken advantage of in only one location, and in two cases lines were read without interning into variables that were commented as interned. We removed the comment and the interning, and changed one use of `==` to `.equals`.

A design flaw relates to the complex interning behavior of the `VarInfoName` class, which represents variable names and formatting, and also their relationships to one another and to program points. All external references to this class are interned (and we verified that all clients treat them properly), but within the class body instances are sometimes interned and sometimes not (for instance, in the middle of a sequence of operations within a method). We discovered locations where uninterned instances could leak to the outside as private fields or as subcomponents of returned references. We have have not yet been able to determine whether this can cause incorrect user-visible behavior. A simpler design would be easier to understand, less error-prone, and likely no less efficient. At the time of our case study, `VarInfoName` was obsolescent: it had been deprecated for over a year and was being retained only for backward compatibility with an older file format.

Our experience so far indicates that the Interned typechecker is easy to use and can be extremely fruitful in identifying errors.

5. Javari checker for mutation errors

Mutation errors are difficult to detect, since the unintended mutation is no different from other mutations that happen throughout the program, and a mutation error is not immediately detected at run time. The Javari [3, 33] type system enables detection and prevention of unintended mutation.

Javari [33] is an extension of the Java language that permits the specification and compile-time verification of immutability constraints. Figure 5 shows the type hierarchy. Programmers can state the mutability and assignability of references using a small set of type annotations.

- The `@ReadOnly` annotation indicates that a reference provides only read-only access; no side effect may be performed through such a reference.
- The `@Mutable` and `@Assignable` annotations exclude parts of an object's state from the mutation guarantee — for example, for a field that is used as a cache.
- The `@QReadOnly` annotation is a mutability wildcard, much like those introduced by `? extends` in Java generics; the "`Q`" in `@QReadOnly` stands for "question mark". This type permits only operations that are allowed for both readonly and mutable types.
- The `@RoMaybe` annotation simulates mutability method overloading, enabling return type mutability to depend on the mutability of parameters. For example, the identity method could be annotated with `@RoMaybe` to indicate that its parameter and return value are either both ReadOnly or both non-ReadOnly.

The plugin issues an error whenever mutation happens through a readonly reference, when non-`@Assignable` fields of a readonly reference are reassigned, or when a readonly expression is assigned to a mutable variable.

Javari's type system has a number of differences from previous immutability proposals; we highlight two of them. 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' abstract state, not just their concrete state.

5.1 Javari case study

We performed case studies on the JOlden benchmark programs[2] [6], a selection of JDK classes, and the Javari checker itself. We converted 55 classes (7756 LOC) of code to Javari by adding annotations such as `@ReadOnly`, using a total of 520 annotations. We also annotated the signatures of 96 library classes and interfaces (4816 LOC), with a total of 1063 annotations. JOlden is written in pre-generics Java, so we added type parameters to it.

The Javari checker found a mutability bug in its own implementation. A global variable containing information about the state of the checker was side-effected when the checker entered an inner class, but was not reset upon exiting. (Our fix allocated a new object instead.) The checker test suite did contain inner classes, but did not contain the right combination of different mutabilities on the outer and inner classes, and additional code after the inner class, to trigger the bug.

The Javari checker issued 3 additional warnings, all of which were false positives issued when a varargs method parameter (`Object... args`) was passed to library method with an annotated array parameter. The current JSR 308 annotation syntax does not permit annotation of the element type, so we could not annotate the argument.

The most difficult part of the case study was annotating undocumented third-party code. It usually lacked documentation regarding side effects. What documentation was

[2] `http://www-ali.cs.umass.edu/DaCapo/benchmarks.html`

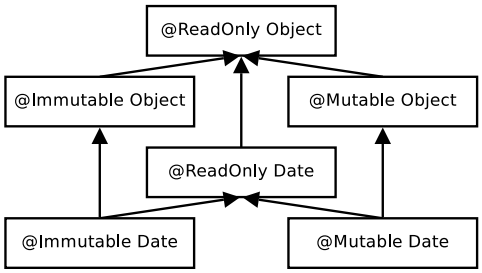

Figure 6: Type hierarchy for IGJ's type qualifiers.

present addressed the receiver, but we were surprised at how many formal parameters were mutated. Annotating our own code was easy and fast.

The most difficult method to annotate was the generic, reflective `Collections.toArray` method. It has a complex specification about which objects get mutated, and about the type of its return value. The incomplete specification did not cause a problem in our case study because of the limited way in which the subject programs used it.

The annotations did not clutter the code, because they appeared mostly on method signatures; leaving local variables unannotated (mutable) was usually sufficient. The few local variable annotations appeared at existing Java casts, where the type qualifier had to be made explicit; flow-sensitive analysis like that of the non-null checker (Section 3) would have eliminated the need for these.

We were able to annotate more local variables in the Javari checker than in the JOlden benchmark, due to better encapsulation and more use of getter methods. Most of the annotations were `@ReadOnly` (288 annotations on classes, 514 annotations on libraries and interfaces). We never used the `@QReadOnly` annotation; the default inherited mutability was expressive enough. We used `@RoMaybe` extensively: on almost every getter method and most constructors, but nowhere else. We used `@Mutable` only 3 times; all 3 uses were in the same class of the Javari visitor, to annotate protected fields that are passed as arguments and mutated during initialization. We used `@Assignable` 16 times, all while annotating a set of inner anonymous classes in JOlden that extended `Enumeration`, that could conceivably be readonly, and that required a reference to the last visited item to be assignable.

6. IGJ checker for mutation errors

Immutability Generic Java (IGJ) [37] is a Java language extension that expresses immutability constraints. Like the Javari language described in Section 5, it is motivated by the fact that a compiler-checked immutability guarantee detects and prevents errors, provides useful documentation, facilitates reasoning, and enables optimizations. IGJ is more powerful than Javari in that it expresses and enforces both reference immutability (only mutable references can mutate an object) and object immutability (an immutable reference points to an immutable object).

Every reference is annotated as `@Immutable`, `@ReadOnly`, `@Mutable` (the default), or `@AssignsFields`; Figure 6 illustrates the relationship among the first three of these.

- A type with an `@Immutable` annotation represents an immutable object, which cannot be mutated via the immutable reference or any aliasing reference.
- A type with a `@ReadOnly` annotation provides only readonly access. No mutation may occur via the reference, but mutation of the referent is possible via an aliasing reference.
- A type with a `@Mutable` annotation represents a mutable object which may be mutated via the reference.
- A method whose receiver type is annotated with `@AssignsFields` is permitted to mutate the receiver in a limited manner, for use in helper procedures called by constructors.
- A field with an `@Assignable` annotation excludes the field from the abstract state and may be reassigned, irrespective of the enclosing object immutability.
- A type with an `@I` annotation simulates mutability overloading; the annotation plays a role similar to that of type variables in Java's generics system.

The plugin issues an error whenever the IGJ type system is violated: in short, mutation through a readonly reference, reassignment of (non-`@Assignable`) fields of a readonly reference, or assignment of an expression to a variable of an incompatible immutability type (e.g. a readonly expression to a mutable variable).

6.1 Annotation IGJ dialect

The original IGJ dialect [37] was not backward-compatible with Java, either syntactically or semantically, and could not specify array mutability. Our checker uses the Annotation IGJ dialect, which corrects these problems.

To express immutability constraints, Annotation IGJ uses (JSR 308) annotations. The original IGJ syntax uses a combination of annotations and generics: each class declaration has a specially-handled type parameter called the immutability parameter.

Annotation IGJ forbids covariant changes in generic type arguments, for backward compatibility with Java. In ordinary Java, types with different generic type arguments, such as `Vector<Integer>` and `Vector<Number>`, have no subtype relationship. The original IGJ dialect permits varying a type argument covariantly in certain circumstances. For example,

```
Vector<Mutable, Integer> <: Vector<ReadOnly, Integer>
                         <: Vector<ReadOnly, Number>
                         <: Vector<ReadOnly, Object>
```

Annotation IGJ supports array immutability. The original IGJ dialect did not permit the (im)mutability of array elements to be specified, because Java's generics syntax does not apply to array elements, or to several other code locations supported by Annotation IGJ.

6.2 IGJ case study

Our preliminary experience suggests that IGJ is useful in expressing and checking important immutability properties.

We performed case studies on the JOlden benchmark programs, the htmlparser library[3], the tinySQL library[4], the SVNKit Subversion client[5], and the IGJ checker itself. In all, we converted 463 classes (128 KLOC with 62,133 possible annotation locations) of code to type-correct IGJ, writing 4,760 annotations. We needed to refactor the code only

[3] `http://htmlparser.sourceforge.net/`
[4] `http://sourceforge.net/projects/tinysql/`
[5] `http://svnkit.com/`

in minor ways noted below. We also annotated the signatures of 201 library classes and interfaces. The IGJ checker issued 26 warnings, which are mostly false positives caused by limitations of our checker implementation: five were at library calls that passed a varargs argument (`(Object... args)`), eight were at assignments to multidimensional arrays, four were at array initializers, and the rest were at instances where the immutability information is lost due to casting to `Object`. (A previous paper [37] presented a preliminary case study that considered less code and used a different IGJ implementation that was hard-coded as javac modifications. Lack of support for array annotations was the biggest problem in the previous case study, particularly because JOlden was transliterated from C and used many arrays.)

Conversion to IGJ revealed representation exposure errors. For example, in the `SVNKit` library, the SSH authentication class constructor takes a char array of the private key and assigns it to a private field without copying; an accessor method also returns that private field without copying. Clients of either method can mutate the array's contents.

Conversion to IGJ also allowed us to find and fix a conceptual problem in several immutable classes, where the constructor left the object in an inconsistent state that was later corrected by another method. This is illegal when the class is immutable, because the second method is not permitted to modify the object. We solved such problems by adding parameters to the constructor/factory to give it access to the complete state of the new object, or by moving all of the logic of object construction into a single method rather than dispersing it.

Conversion to IGJ revealed an unusual design pattern in SVNKit: some getters have side effects, and some setters have none! For example, `getSlotsTable` is actually a factory method that returns the same `SlotsTable` object on each invocation, but side-effects that object according to the argument to `getSlotsTable`. Method `setPath` is also a factory method that returns a new `SVNURL` object like the receiver, but with one field set to a different value. Documenting these unexpected mutation facts (about both arguments and results) made the code much more comprehensible.

Annotating the IGJ checker made the code easier to understand, because the annotations distinguished among the uses of unmodifiable and modifiable collections.

We were able to use both immutable classes and immutable objects. SVNKit used the latter for `Date` objects that represent the beginning and expiration of file locks; the URL to the repository (IGJ could simplify the current design, which uses an immutable `SVNURL` class with setter methods that return new instances); and many `Lists` and `Arrays` of metadata. IGJ could also permit use of immutable objects in some places where immutable classes are currently used, increasing flexibility.

Most fields used the containing class's immutability parameter. We used few mutable fields; one of the rare exceptions was a collection (in `SVNErrorCode`) that contains all `SVNErrorCodes` ever created. We used `@Assignable` fields only 5 times — to allow the receiver of a tree rebalancing operation, or the receiver of a method that resizes a buffer without mutating the contents, to be marked as `ReadOnly`. In (only) one case, we would have liked multiple immutability parameters for an object: the return value of `Map.keySet` allows removal but disallows insertion.

Some classes are really collections of methods, rather than representing a value as the object-oriented design paradigm dictates. We found immutability types a poor fit to such classes, but leaving them unannotated (the default is mutable, for backward compatibility with Java) worked well.

To our surprise, we found we preferred Annotation IGJ to the original IGJ dialect, despite the fact that IGJ is generics-inspired and that its type system is explained in terms of generics. The original IGJ dialect mixes immutability and generic arguments in the same `<...>` list, and the immutability parameter comes first but may be omitted. Thus, a programmer must stop to interpret each list of type arguments. The new dialect was somewhat easier to write as well, and it was easier to convert back to ordinary Java for processing with a standard Java compiler. Java namespace limitations force the original IGJ dialect to use a different name for method annotations than for immutability parameters. Java generally uses prefix modifiers, so the suffix-like form `List<ReadOnly, Date<Mutable>>` felt less natural to us than `@ReadOnly List<@Mutable Date>`.

Annotating existing code is an important test of IGJ, but IGJ is likely to be even more effective on code that is designed with immutability in mind. We saw many places that a different — and better! — design would have been encouraged by IGJ. This was characteristic of each of our case studies. We look forward to the pleasant experience of programming with the benefit of these checkers from the beginning of a project.

7. Related work

There is a rich literature discussing the theoretical underpinnings of qualified type systems; a good place to start is Foster et al.'s classic paper [20]. That paper presents type inference algorithms, but our focus is on expressing and checking types. Another difference is that in addition to "negative" and "positive" qualifiers that induce subtypes and supertypes, we are interested in more complex type systems that may introduce additional rules and constraints.

Bracha's [5] term "pluggable type system" means a dynamically typed system in which all static type-checking is optional and has no effect on runtime semantics, and all type-checker messages are warnings rather than errors. A programmer can choose to use as many or as few type-checkers as desired, depending on her needs. The latter statement applies to our system as well, but not the former. For reasons of backward compatibility, a program is required to type-check in the ordinary Java type system when the qualifiers are ignored. Rather than replacing the existing type system, a compiler plug-in adds additional constraints and makes additional guarantees.

Andreae et al.'s JavaCOP system [2] shares the same goal as our work: creation of a practical framework for implementing pluggable type systems [5] in Java, starting from Java 5 annotations. JavaCOP provides a pattern-matching language for specifying type rules for Java constructs; a type system designer create a type-checker using a combination of these declarative rules and procedural Java code. For instance, to specify that a qualified type is a subtype of an unqualified type, the designer would write separate rules for assignments, method overriding, etc., and auxiliary Java methods that the rules call out to. Andreae et al. [2] provide an impressive collection of a dozen checkers that can be written with their system. However, these checkers have

been run only on toy programs; lack of flow-sensitivity in the non-null analysis, incomplete implementations (both missing rules for certain Java constructs and also simplified type systems), and other problems prevent practical use of the checkers. By contrast, we have created only four checkers to date, but have demonstrated their utility on real Java programs. Our system integrates with standard tools such as the Java Tree API and the JSR 269 annotation-processing system, whereas the JavaCOP authors chose to implement their own incompatible variants. Our system incorporates an extension to Java's annotation system that permits specification of qualified types. Our system does not require use of a custom Java compiler itself (though it does require a compiler that supports the new annotation syntax, which is intended to become a standard part of the Java 7 language). Additionally, our system is publicly available, whereas JavaCOP is not (as of Sept. 2007).

JQual [24] is another tool that adds user-defined type qualifiers to Java. Our focus is type checking, but JQual addresses the more difficult type inference problem by translating the ideas of earlier CQual [21] research to the object-oriented context: JQual generates type constraints from syntax-directed rules, then solves them to produce a new typing of the program. JQual does not handle generic types, but it does permit programmers to enable field-sensitivity on a field-by-field basis (enabling it globally is not scalable) as a stand-in. JQual also operates context-sensitively, similar to the `ROMaybe` qualifier of Javari. JQual has been used in two case studies: to identify the enums and addresses that are part of a public JNI interface, and to infer types for a Javari-like language (see below).

Fong [19] describes a framework for implementing pluggable type systems (more precisely, verification modules) for Java bytecodes. These are implemented by the classloader and can replace or augment the standard bytecode verifier. By contrast, our work focuses on source-code checking. A byte-code verifier could augment a source-code checker.

Null pointer errors are a bugaboo of programmers, and significant effort has been devoted to tools that can eradicating them. Engelen [12] ran a significant number of null-checking tools and reports on their strengths and weaknesses; Chalin [7] gives another recent survey. We mention four notable practical tools. ESC/Java [18] is a static checker for null pointer dereferences, array bounds overruns, and other errors. It translates the program to the language of the Simplify theorem prover [9]. This is more powerful than a type system, but suffers from scalability limitations. The JastAdd extensible Java compiler [10] includes a module for checking and inferencing of non-null types [11]. To handle manipulation of partially-initialized objects, JastAdd implements a raw type system [16], which increases the number of safe dereferences in the program from 69% to 71%. The JACK Java Annotation CheckKer [27] is similar to JastAdd in that both use flow-sensitivity and a raw type system and have been applied to nontrivial programs. Unlike JastAdd but like our tool, JACK is a checker rather than an inference system. The null pointer bug module of FindBugs [25] takes a very different approach to the other work (and our own). Rather than trying to prove the absence of errors via an analysis that is as precise as practical, FindBugs assumes that many errors exist and aims to find a few of them. Like the inference systems, FindBugs requires only a few user annotations. FindBugs uses an extremely coarse analysis that yields mostly false positives — it would indicate that most dereferences are of possibly-null values. Then, FindBugs uses heuristics to discard reports about values that might result from infeasible paths, flow through a catch clause, are returned by a method invocation, etc.

Interning (use of a canonical representation) has been used since at least the 1950s; Ershov [15] discusses checking for duplicate formulas in an arithmetic optimizer. Interning has been widely used in Lisp data structures [23, 1], where the name "hash consing" referred to the *cons*truction of objects making use of a *hash* table. More recently, Vaziri et al. [34] give a declarative syntax for specifying the interning pattern in Java and found equality-checking and hash code bugs similar to ours; they use the term "relation type" for an interned class. Marinov and O'Callahan [28] describe a dynamic analysis that identifies interning and related optimization opportunities. Based on the results, the authors then manually applied interning to two SpecJVM benchmarks, achieving space savings of 38% and 47%. A more representative example is the Eiffel compiler; interning strings resulted in a 10% speedup and 14% memory savings [36]. Our system is more flexible than previous interning approaches [28, 17, 34] in that it neither requires all objects of a given type to be interned nor makes interned objects type-incomparable with (not a subtype of) uninterned ones.

Our implementation is the first for the Javari language [33], and incorporates several improvements that are described in a technical report [32]. Birka [3] implemented a syntactic variant of the Javari2004 language, which is a different design than the current Javari language. JavaCOP [2] and JQual [24] have been used to implement subsets of Javari that do not handle method overriding, omitting fields from the abstract state, templating, generics (in the case of JQual), and other features that we have found important for practical use. JavaCOP's implementation was never executed on large programs, and whereas JQual's was, the JQual inference results were correct for only 35 out of the 50 variables that the authors examined by hand.

Our implementation is the second for IGJ, but (as described in Section 6) our Annotation IGJ dialect is more practical and permits more complete annotation.

Two of our checkers have been described (in a few paragraphs) in OOPSLA posters [29, 26], but no previous published description has appeared.

8. Conclusion

This paper has presented a framework for creating and checking custom Java type qualifiers. Type system designers extend the framework to create type-checking compiler plugins, programmers add annotations to their programs, and the Java compiler reports type system violations at compile-time. Our framework builds atop existing Java tools and APIs and on JSR 308, providing a type qualifier framework that is compatible with the Java language, virtual machine, and toolchain.

To evaluate our framework, we defined checkers for four type systems — non-null, interned, Javari, and IGJ — and performed case studies in which these checkers revealed previously unknown errors in real programs, or verified the absence of such errors. We believe that our framework is the first practical system for expressing and enforcing pluggable type systems in the Java language. The tools described in this paper are publicly available for download from

http://pag.csail.mit.edu/jsr308/.

9. References

[1] John R. Allen. *Anatomy of LISP*. McGraw-Hill, New York, 1978.

[2] Chris Andreae, James Noble, Shane Markstrum, and Todd Millstein. A framework for implementing pluggable type systems. In *OOPSLA*, pages 57–74, Oct. 2006.

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

[4] Gilad Bracha. JSR 175: A metadata facility for the Java programming language. http://jcp.org/en/jsr/detail?id=175, Sep. 30, 2004.

[5] Gilad Bracha. Pluggable type systems. In *OOPSLA Workshop on Revival of Dynamic Languages*, Oct. 2004.

[6] Brendon Cahoon and Kathryn S. McKinley. Data flow analysis for software prefetching linked data structures in Java. In *PACT*, pages 280–291, Sep. 2001.

[7] Patrice Chalin and Perry R. James. Non-null references by default in Java: Alleviating the nullity annotation burden. In *ECOOP*, pages 227–247, Aug. 2007.

[8] Joe Darcy. JSR 269: Pluggable annotation processing API. http://jcp.org/en/jsr/detail?id=269, May 17, 2006. Public review version.

[9] David Detlefs, Greg Nelson, and James B. Saxe. Simplify: A theorem prover for program checking. Technical Report HPL-2003-148, HP Labs, Palo Alto, CA, July 23, 2003.

[10] Torbjörn Ekman and Görel Hedin. The JastAdd extensible Java compiler. In *OOPSLA*, Oct. 2007.

[11] Torbjörn Ekman and Görel Hedin. Pluggable checking and inferencing of non-null types for Java. *Journal of Object Technology*, 2007.

[12] Arnout F. M. Engelen. Nullness analysis of Java source code. Master's thesis, University of Nijmegen Dept. of Computer Science, Aug. 10 2006.

[13] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE TSE*, 27(2):99–123, Feb. 2001.

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

[15] A. P. Ershov. On programming of arithmetic operations. *CACM*, 1(8):3–6, Aug. 1958.

[16] Manuel Fähndrich and K. Rustan M. Leino. Declaring and checking non-null types in an object-oriented language. In *OOPSLA*, pages 302–312, Nov. 2003.

[17] Jean-Christophe Filliâtre and Sylvain Conchon. Type-safe modular hash-consing. In *ML*, pages 12–19, Sep. 2006.

[18] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *PLDI*, pages 234–245, June 2002.

[19] Philip W. L. Fong. Pluggable verification modules: An extensible protection mechanism for the JVM. In *OOPSLA*, pages 404–418, Oct. 2004.

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

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

[22] Erich Gamma, Richard Helm, Ralph E. Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley, Reading, MA, 1995.

[23] E. Goto. Monocopy and associative algorithms in an extended Lisp. Technical Report 74-03, Information Science Laboratory, University of Tokyo, Tokyo, Japan, May 1974.

[24] David Greenfieldboyce and Jeffrey S. Foster. Type qualifier inference for Java. In *OOPSLA*, Oct. 2007.

[25] David Hovemeyer, Jaime Spacco, and William Pugh. Evaluating and tuning a static analysis to find null pointer bugs. In *PASTE*, pages 13–19, Sep. 2005.

[26] Telmo Luis Correa Jr., Jaime Quinonez, and Michael D. Ernst. Tools for enforcing and inferring reference immutability in Java. In *OOPSLA Companion*, Oct. 2007.

[27] Chris Male and David J. Pearce. Non-null type inference with type aliasing for Java. http://www.mcs.vuw.ac.nz/~djp/files/MP07.pdf, Aug. 20, 2007.

[28] Darko Marinov and Robert O'Callahan. Object equality profiling. In *OOPSLA*, pages 313–325, Nov. 2003.

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

[30] Jeff H. Perkins and Michael D. Ernst. Efficient incremental algorithms for dynamic detection of likely invariants. In *FSE*, pages 23–32, Nov. 2004.

[31] Frank Pfenning. Dependent types in logic programming. In Frank Pfenning, editor, *Types in Logic Programming*, chapter 10, pages 285–311. MIT Press, Cambridge, MA, 1992.

[32] Matthew S. Tschantz. Javari: Adding reference immutability to Java. Technical Report MIT-CSAIL-TR-2006-059, MIT CSAIL, Sep. 5, 2006.

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

[34] Mandana Vaziri, Frank Tip, Stephen Fink, and Julian Dolby. Declarative object identity using relation types. In *ECOOP*, Aug. 2007.

[35] Peter von der Ahe. JSR 199: Java compiler API. http://jcp.org/en/jsr/detail?id=199, Dec. 11, 2006.

[36] Olivier Zendra and Dominique Colnet. Towards safer aliasing with the Eiffel language. In *IWAOOS*, pages 153–154, June 1999.

[37] Yoav Zibin, Alex Potanin, Mahmood Ali, Shay Artzi, Adam Kieżun, and Michael D. Ernst. Object and reference immutability using Java generics. In *ESEC/FSE*, Sep. 2007.