

# Type Qualifier Inference for Java

David Greenfieldboyce

University of Maryland, College Park  
dgreenfi@cs.umd.edu

Jeffrey S. Foster

University of Maryland, College Park  
jfoster@cs.umd.edu

## Abstract

Java’s type system provides programmers with strong guarantees of type and memory safety, but there are many important properties not captured by standard Java types. We describe JQual, a tool that adds user-defined *type qualifiers* to Java, allowing programmers to quickly and easily incorporate extra lightweight, application-specific type checking into their programs. JQual provides type qualifier inference, so that programmers need only add a few key qualifier annotations to their program, and then JQual infers any remaining qualifiers and checks their consistency. We explore two applications of JQual. First, we introduce opaque and enum qualifiers to track C pointers and enumerations that flow through Java code via the JNI. In our benchmarks we found that these C values are treated correctly, but there are some places where a client could potentially violate safety. Second, we introduce a readonly qualifier for annotating references that cannot be used to modify the objects they refer to. We found that JQual is able to automatically infer readonly in many places on method signatures. These results suggest that type qualifiers and type qualifier inference are a useful addition to Java.

**Categories and Subject Descriptors** D.2.4 [Software Engineering]: Software/Program Verification—Validation; D.3.2 [Programming Languages]: Language Classifications—Object-oriented languages; F.3.2 [Logics and Meanings of Programs]: Semantics of Programming Languages—Program analysis

**General Terms** Languages, Verification

**Keywords** JQual, Java, type qualifiers, readonly, mutable, opaque, transparent, tracked, context-sensitivity, field-sensitivity, context-free language reachability

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

OOPSLA’07, October 21–25, 2007, Montréal, Québec, Canada.  
Copyright © 2007 ACM 978-1-59593-786-5/07/0010...\$5.00.

## 1. Introduction

The Java programming language has a strong type system that can be used to enforce useful program properties. However, many important properties can be hard to encode as standard types, and it may be difficult to incorporate new properties into the type hierarchy of an existing program.

To address this problem we present JQual, a tool for inferring user-defined *type qualifiers* in Java. A type qualifier is an atomic property that refines a standard type. For example, we have used JQual to add opaque, a new type qualifier, to programs that use the JNI. The opaque qualifier annotates Java ints that actually represent C pointer values. Using JQual, we can enforce the integrity property that ordinary Java integers, which we qualify with transparent, are never passed to opaque positions, since then they might mistakenly be treated as pointers and dereferenced. This example is a fairly typical use of type qualifiers, in which the programmer knows some extra properties about certain values, but those values are not distinguished by the standard types.

In JQual, type qualifiers can be applied to any type. For example, we also used JQual to infer a readonly qualifier, based on Javari [8, 47]. For any class C, a reference of type readonly C may not be used to modify the object it refers to, which is a particularly useful annotation for method arguments and results. One interesting feature of readonly is that it is “sticky” across field access: If x is readonly, then so is x.f. Thus type qualifiers can propagate in ways that ordinary types do not, in this case from a reference to the object to which it refers.

When users specify their type qualifiers to JQual, they supply a *subtyping* order that relates sets of qualifiers. For example, mutable references, which are ordinary, writable Java references, can be passed to readonly positions, but not vice-versa, and so mutable < readonly. The addition of subtyping makes qualifier inference more flexible, and supporting subtyping on qualifiers in a language with subclassing seems very natural.

The key feature of JQual is *type qualifier inference*, which allows programmers to add a few qualifier annotations to their program, and then JQual infers the remaining qualifiers and checks their consistency. We formalize type qualifier inference as a system called Core JQual, which operates

on a variant of Featherweight Java [22] with optional type qualifier annotations.

Core JQual is *context-insensitive* and *field-based*, meaning all instances of a class share the same field types. We extend Core JQual to FS JQual, which adds *field-sensitivity*, in which each instance of an object has its own field types. The key feature of FS JQual is that field-sensitivity is selective—the programmer specifies which fields should be treated field-sensitively, and which fields should not. This greatly improves performance over a full field-sensitive analysis, and we have generally found it is easy to identify which fields should be made field-sensitive. We further extend our formalism to CS/FS JQual, which supports both field- and context-sensitivity. We introduce context-sensitivity by encoding it as a *context-free language reachability* problem on a constraint graph [36, 38]. Context-sensitivity is especially important for a field-sensitive analysis, so that object types are not conflated by common constructors and methods.

We have implemented JQual as an Eclipse plug-in [17], and we used JQual for the two applications mentioned above. First, we applied opaque inference to a number of libraries that use the JNI, using a simple C analysis to add the necessary opaque qualifiers to native method signatures. For this analysis, we also inferred  $\text{enum}_i$  qualifiers, which mark integer values in the range  $[0..i]$ , and which ensure that only integers in the expected range are passed to C enums. We found that context- and field-sensitivity were critical in reducing the warning counts to reasonable levels. Across our benchmarks, most opaque and all enum-qualified integers were used safely, but we found several places where opaque integers are exposed outside the library code, and hence could be compromised by careless clients.

Second, we applied readonly inference to a variety of Java programs, including SPEC benchmarks [2] and programs downloaded from SourceForge [1]. We also inferred final annotations on these programs at the same time. Using context-insensitive, field-based analysis, JQual inferred that 48% of non-primitive method arguments and results are readonly, and 29% of fields are final. Added field-sensitivity for library container classes and context-sensitivity increased readonly to 62% of the positions and final to 42% of the fields. Overall, JQual was able to infer many interesting uses of readonly across the benchmarks.

In summary, the main contributions of this work are:

- We introduce opaque and transparent qualifiers to distinguish integers that must be treated abstractly from ordinary Java integers, and  $\text{enum}_i$  qualifiers to track allowed ranges of integers. We also introduce readonly and mutable qualifiers for Java to add reference immutability, based on Javari [8, 47]. (Section 2)
- We present Core JQual, a formal type qualifier inference system for a simple object-oriented languages. We show how to extend this system to FS JQual, which includes

field-sensitivity, and to CS/FS JQual, which adds context-sensitivity. (Section 3)

- We describe JQual, our implementation for Java, and use JQual in our two applications: Inferring opaque, transparent, and  $\text{enum}_i$  qualifiers in code that uses the JNI, and inferring readonly in a range of Java programs. We found that JQual was able to find several potential opaque violations in our benchmarks, as well as many occurrences of readonly and final. (Section 4)

In prior work, we described CQual, a type qualifier system for C with many applications [4, 9, 14, 15, 16, 18, 24, 42, 49]. In the current work, we show how to incorporate type qualifiers into Java, and explore novel applications. Our results suggest that type qualifiers and type qualifier inference are a useful framework for adding lightweight, application-specific checking to Java.

## 2. Applications

Before presenting our type qualifier inference system formally, we discuss our two major applications of JQual in more depth, and sketch several other potential applications.

### 2.1 Enhanced Type Checking for the JNI

The first application we explored is adding enhanced type checking for the Java Native Interface (JNI). The JNI allows Java code to call functions written in C, which is extremely useful but potentially unsafe. One particular problem can occur when a C program passes pointer values to Java. For example, we examined `libgtk-java` [23], a JNI windowing toolkit whose interface can be used to create various GUI entities, e.g., windows or buttons. Pointers to those objects are sent to Java as ints, so that the programmer can pass them back to the JNI library to manipulate the window components. However, since the ints are just ordinary integers as far as Java is concerned, the programmer could inadvertently pass bogus pointer values back to C, thereby potentially causing memory corruption.

We can prevent this problem using type qualifiers. We introduce a qualifier `opaque` to mark integers that are treated as pointers in C and a qualifier `transparent` to mark integers that have been manipulated or created inside of Java. We want to forbid transparent integers from being used in opaque positions, so `transparent`  $\not\prec$  `opaque` in the ordering on these qualifiers. On the other hand, it is safe to allow Java to manipulate opaque values it receives from C (e.g., by printing them), as long as they are never passed back to C. Thus the final qualifier ordering is `opaque`  $<$  `transparent`, so that opaque values can be used in transparent positions, but not vice-versa.

While this application is specialized for the JNI, the basic idea is quite general: We wish to enforce an integrity property, namely that transparent data is not used in opaque positions. An analogous example is enforcing the security

```

1 class GUILib {
2     public native static opaque int makeWin(enum2 int t);
3     public native static void setFocus(opaque int windowPtr);
4 }

5 struct window { ... };
6 enum windowType { a = 0, b, c };
7 jint Java_GUILib_makeWin(enum windowType t) {
8     struct window *w = make_window(t);
9     return (int) w;
10 }
11 void Java_GUILib_setFocus(int jwindow) {
12     struct window *w = (struct window *) jwindow;
13     set_focus(w);
14 }

```

(a) Interface code to C library

```

15 class Client {
16     public void m() {
17         opaque int w = GUILib.makeWin(1);
18         // ok - 1 is enum1, which is ≤ enum2
19         opaque int v = GUILib.makeWin(4);
20         // error - 4 is enum4, which is ≱ enum2
21         setFocus(w); // ok - opaque int passed in
22         setFocus(42); // error - created ints are transparent
23         setFocus(w + 3); // error - manipulated ints are transparent
24         transparent int t = w; // ok - opaque treated as transparent
25         t++; // ok - t is transparent
26         setFocus(t); // error - transparent not opaque
27     }
28 }

```

(b) Java client of GUILib

**Figure 1.** Examples of JNI qualifier usage

property that tainted (i.e., untrusted) data is never used in untainted (i.e., trusted) positions [42].

In addition to passing C pointers across the JNI, we found that native C code sometimes accepts integers from Java that are treated as `enums` in C, meaning they should be within a certain range. For example, an `enum` parameter might be used to select different attributes of a GUI widget. In this case, Java code could pass an out-of-range value to C, and although the C compiler itself may not complain about this kind of problem, we can detect it with type qualifiers.

We assume that enumerations are contiguous and start from 0. Then for an `enum` with maximum value  $i$ , we introduce a new qualifier `enumi`, which we also assign to occurrences of the integer  $i$  in Java. We add the subtype ordering `enumi < enumj` for  $i < j$ , since the range  $[0..i]$  is included in the range  $[0..j]$  for  $i < j$ . As before, our application is specialized to the JNI, but this basic idea could be applied to enforce similar range-checking properties.

Figure 1 shows an example of using `opaque`, `transparent`, and `enum` qualifiers. Part (a) shows the “glue code” that connects Java to a hypothetical C library. Lines 1–4 define a class `GUILib` with two native methods, `makeWin` and `setFocus`. The C code for these methods is shown on lines 5–14. Line 5 declares the C type `struct window`, and line 6 declares `enum windowType` containing values 0–2.

Then lines 7–9 declare the C function corresponding to the native `makeWin` method—note the mangled function name, which is part of the JNI specification [28]. (We have hidden some other details of using the JNI for clarity.) Because the C function is declared to take an `enum windowType`, the Java native method on line 2 takes an `enum2 int` as a parameter, to specify the three possible permitted values. Then on lines 8–9, we see that a pointer type is passed back to Java, and hence on line 2, the return type of the method is marked as `opaque`. Similarly, lines 11–14 define the C function corresponding to the `setFocus` method. The function takes an integer as an argument and casts it to a pointer, and hence `setFocus`’s parameter is also `opaque` on line 3.

Figure 1(b) shows sample client code that uses `GUILib`’s methods correctly and incorrectly. On line 17, we call `makeWin` to create a new window, passing argument 1. This is allowed because 1 has the qualifier `enum1`, which is compatible with the qualifier `enum2` of the formal parameter. The return value of `makeWin` is stored in an `opaque` integer, since it represents a C pointer. On line 19, the call to `makeWin` is forbidden, because we try to pass in an integer that is out of range. Next, on line 21, we pass `w` to `setFocus`, which is allowed because `w` is `opaque`. On the other hand, we may not pass `setFocus` an integer created (line 22) or manipulated (line 23) by Java. We are allowed to copy an `opaque` integer to a `transparent` integer, but since we then may manipulate it (line 25), we cannot pass `transparent` integers back to C (line 26).

In our experiments (Section 4.1), we found that C pointers and enumerations are generally treated safely in Java code, but there are some places where insufficient encapsulation allows opportunities for library clients to pass Java integers to `opaque` positions.

## 2.2 Inferring Immutability Properties

The second application we explored is inferring immutability in Java. Immutability, the guarantee that particular memory locations will not be updated, is a useful property that can make programs easier to understand. For example, a caller might want to know whether passing an object to a method could cause that object to change. In Java, the programmer can use `final` to mark fields whose values cannot be changed,<sup>1</sup> but this enforces only one aspect of immutability. Ernst et al have proposed a language extension called `Javari` [8, 47] that adds *reference immutability* to Java. In `Javari`, a variable marked with the qualifier `readonly` cannot be used to write to fields of the object it refers to. Ordinary variables without this restriction have the qualifier `mutable`.

We can use `JQual` to infer `Javari`-style qualifiers in Java and, at the same time, infer `final`. There are some differences between our qualifiers and `Javari`, which we discuss below.

<sup>1</sup> The keyword `final` can also annotate classes and methods, in which case it prevents subclassing and overriding, respectively. We ignore these uses of `final` in this paper.

```

1 class C {
2   nonfinal mutable C c;
3   nonfinal int x;
4 }
5 class D {
6   void foo() {
7     final mutable C f = new C();
8     f.x = 17; // ok - writes field of final variable
9     f = new C(); // error - assigns to final variable
10
11    nonfinal readonly C r = new C();
12    r = new C(); // ok - assigns to nonfinal
13    r.x = 17; // error - writes field of readonly reference
14    r.c.x = 17; // error - writes field of readonly reference
15
16    nonfinal mutable C m;
17    m = r; // error - assigns readonly to mutable
18    m.x = 17; // ... which would allow write to field
19    r = m; // ok - may assign mutable to readonly
20  }
21 }
22 class E {
23   nonfinal int x;
24   void bar() mutable { // bar() is mutable...
25     x = 17; // ... because it modifies a field of this
26   }
27   void baz() {
28     nonfinal readonly r = new C();
29     r.bar(); // error - calls mutable method of readonly ref.
30   }
31   nonfinal readonly C f;
32   public E() {
33     f.x = 17; // ok - may write to readonly field in constructor
34   }
35 }

```

**Figure 2.** Examples of immutability qualifier usage

We introduce four qualifiers for this application: `readonly`, `mutable`, `final`, and `nonfinal`, where the last qualifier explicitly marks variables and fields that are not final. We illustrate the desired behavior for these qualifiers in Figure 2.

Lines 1–4 define a class `C` with two fields, `c` and `x`. Both fields are explicitly annotated with `nonfinal`, which in Java would normally be denoted by the absence of `final`. The field `c` is also mutable. Note that it would not make sense to mark `x` as mutable, because it contains a primitive.

The next part of the figure shows a class `D` with a method `foo`. On line 7, we create a new instance of `C` and store it in a final, mutable variable `f`. Since `f` is mutable, we may write through it on line 8 to modify one of its fields. However, the assignment on line 9 is forbidden, because it would modify `f` itself, which is final. Notice that `final` and `nonfinal` are properties of the location of field `f`, and thus we say that `final` and `nonfinal` are *reference level qualifiers*. In particular, they specify whether we can modify what is stored in `f`, but they do not place any constraints on what we can do with the contents of `f` (e.g., we can freely write through it). This contrasts with the qualifiers used in the JNI application, which all refine the types of values (in particular, integers), and so are *value level qualifiers*.

Next on line 11, we initialize `r`, which is `nonfinal` and `readonly`. Therefore on line 12, we may modify `r` because it is `nonfinal`, but on lines 13 and 14 we may not write to any of `r`'s fields, either directly or transitively. Notice that `readonly` and `mutable` are properties of the contents of `r`, and not of the location `r` itself, and thus `readonly` and `mutable` are value level qualifiers.

Line 17 tries to write a `readonly` variable to the mutable variable `m` declared on line 16. This must be forbidden, because otherwise on line 18 we could write through `m` to change a field of `r`, which would break the property that starting from `r` we can never write to any of the fields. On the other hand, as line 19 shows, we may assign a mutable variable to a `readonly` variable, because strictly fewer operations are permitted on `readonly` variables. Putting these together, we choose the qualifier order `mutable < readonly`. There is no need to choose an order among `final` and `nonfinal`, since field locations are not first-class values in Java.

Finally, the last part of Figure 2 shows a class `E` that illustrates some other features of `readonly` and `mutable`. Just as variables and fields may be qualified, so too may be arguments and return values. Moreover, we allow qualifiers on this, written after the method. Lines 24–26 show an example, in which `bar` is marked as `mutable` because it writes to a field `this`. Therefore on line 29, we may not invoke `r.bar()`, because `r` was declared `readonly` on line 28. Only `readonly` methods may be invoked on `readonly` objects. Following Javari, we do allow `readonly` to be violated in constructors. On line 33, the write through `readonly` field `f` is permitted. We also allow writes to final fields in constructors. (Java technically allows only one write to a final field, whereas in JQual we permit multiple writes.)

**Differences from Javari** As this example suggests, these four qualifiers provide a concise but rich way of specifying immutability properties of variables and fields. Our version of these qualifiers is slightly different than Ernst's Javari proposal [47], which they are modeled after. In Javari, classes may also be marked `readonly` to indicate that all of their instances are as well, and JQual supports but does not infer this behavior. Javari allows the programmer to explicitly add generic immutability annotations on classes. We support field- and context-sensitivity during inference (Sections 3.2 and 3.3), which are equivalent, but currently do not include source-level notation for them.

Javari also allows somewhat finer control of the writeability of fields. In JQual, a `readonly` reference cannot be used to modify any fields of the object it refers to, while in Javari, this behavior can be adjusted on a field-by-field basis.

One of the more important differences between JQual and Javari is in method subtyping. Javari does not allow method overriding on the basis of immutability. Instead, a change to mutability between the method of a supertype and the method of a subtype causes the method to be overloaded. In contrast, JQual ignores qualifiers when determining over-

riding versus overloading, since deciding between the two while inferring qualifiers would be difficult. JQual therefore allows qualifiers to be in a subtyping relationship among overriding and overridden methods.

Lastly, Javari inserts runtime checks to allow for safe downcasts and reflection. This is beyond the scope of the current work on JQual, which focuses on static analysis rather than code transformation, and so we track qualifiers through casts and ignore calls to the reflection API.

### 2.3 Other Potential Applications

We believe type qualifier inference for Java has a wide variety of applications. In general, JQual is effective for *source-sink* problems, which involve tracking the flow of data from a set of sources to a set of sinks. To use qualifier inference for such problems, the programmer only needs to add qualifier annotations to the sources and sinks, and then qualifier inference determines the intermediate qualifiers in the rest of the program. Assuming there are relatively few sources and sinks, the annotation burden for using qualifiers can be quite minimal. Also, for the applications in this paper, we use JQual as a whole program analysis. However, since it is based on types, JQual can also be used to analyze a module in isolation, as long as qualifier information is provided at the module boundary. We discuss several potential applications of JQual briefly.

Earlier we mentioned that we could use JQual to track tainted data through Java and ensure it does not reach untainted positions. This analysis could be especially useful for Java-based web applications [21], which are vulnerable to attacks using unchecked inputs, including SQL injection queries, cross-site scripting, cookie poisoning, and reflection injection [30]. For example, we could mark as tainted the return values of methods that receive input from users, such as those in the `HttpServletRequest` class [3], and then annotate appropriate positions with untainted.

Another potential application of type qualifiers is to distinguish among different kinds of Strings. For example, we could use a `url` qualifier to annotate strings that are URL-encoded, an `html` qualifier for strings that represent valid HTML, or an `sql` qualifier for strings that are supposed to be SQL queries. JQual would not by itself verify that the strings have valid contents, but it can ensure that they are passed to and from methods that expect strings of those types. Similarly, byte arrays could be qualified with `utf8`, `utf16`, or similar to denote the encoding of their contents, and then JQual could ensure they are manipulated by the correct methods.

Lastly, JSR 305 proposes to develop standard annotations for Java programs to allow tools to check a variety of correctness properties [35]. Suggested standard annotations include ones to specify nullness, tainting, concurrency properties, and internationalization. JQual may be able to check and infer several of these properties.

$$\begin{aligned}
 P &::= L^* \\
 L &::= \text{class } C \text{ extends } D \{F_1; \dots; M_1; \dots\} \\
 T &::= C \mid C^Q \\
 F &::= T f \\
 M &::= T_0 m(T_1 x_1, \dots, T_n x_n) T \{e\} \\
 e &::= x \mid \text{null} \mid e_1; e_2 \mid e.f \mid e_1.f := e_2 \\
 &\quad \mid e.m(e_1, \dots, e_n) \mid (T) e \mid \text{new } T \\
 C &::= \text{Object} \mid \langle \text{class names} \rangle \\
 Q &::= \kappa \mid \langle \text{qualifier constants} \rangle \\
 x &::= \text{this} \mid \langle \text{variable names} \rangle \\
 f &::= \langle \text{field names} \rangle \\
 m &::= \langle \text{method names} \rangle
 \end{aligned}$$

Figure 3. Java-like source language

## 3. Type Qualifier Inference

Next we present JQual’s type inference system formally. We present three type systems: *Core JQual*, which is a field-based, context-insensitive inference system, *FS JQual*, a field-sensitive extension, and *CS/FS JQual*, which further adds context-sensitivity.

We describe both systems for the source language in Figure 3, which is a variation of Featherweight Java [22] that has been extended with qualifiers. Programs  $P$  consists of a sequence of class definitions  $L$ . Each class extends exactly one other class, and there is a built-in base class `Object`. A class definition contains a sequence of field declarations  $F$  and method declarations  $M$ . Types  $T$  that appear in the source code may either be ordinary Java types  $C$  or *qualified types*  $C^Q$ . A qualifier  $Q$  is either a *qualifier variable*  $\kappa$ , which is an unknown that must be solved for, or a *qualifier constant* such as `readonly` or `opaque`. We assume the set of qualifier constants is fixed in advance, and without loss of generality we allow only one qualifier constant per type [15]. Qualifier variables do not appear in the program text.

Field declarations contain a type  $T$ , and method declarations may also contain qualified types for their arguments and returns. In a method declaration, the type  $T$  that appears just after the parameter list is the type of `this` within the method, which is our mechanism for supplying a qualifier on this.

When a method is invoked, it evaluates to its body, which is an expression  $e$ . Expressions include variables  $x$  (either `this` or a parameter name), the special value `null`, sequencing  $e_1; e_2$ , field access  $e.f$ , field assignment  $e_1.f := e_2$ , and method invocation  $e.m(e_1, \dots, e_n)$ . We also allow type casts  $(T) e$  on expressions. We assume that the class in such a cast is checked at run time, but the qualifier is not. Finally, the expression `new T` creates a new instance of class described by  $T$ , with its fields initialized to `null`. Note that unlike Featherweight Java, we do not include constructors or calls to `super` in our language.



method types  $\mu$ , respectively. Core JQual is field-based because field types are stored in the global  $CT$ , no matter what instance they come from.

Field types have the form  $ref^Q(\tau)$ . Here  $\tau$  is the contents type and includes the value-level qualifier, and  $Q$  is the reference-level qualifier on the field itself. For example, a final mutable field has type  $ref^{final}(C^{mutable})$ . Method types have the form  $(\tau_1 \times \dots \times \tau_n) \tau \rightarrow \tau_0$ . Here  $\tau_0$  is the return type, the other  $\tau_i$  are the argument types, and  $\tau$  type of this within the method. As a shorthand, we write  $o(f_i)$  to mean the  $\phi_i$  corresponding to  $f_i$  in  $\tau$ , and similarly for  $o(m_j)$ .

To perform inference, JQual needs to translate source types  $T$ , which may or may not contain qualifiers, into inference types. Figure 4(b) defines this translation. The first two lines define the function  $fresh(T)$ , which qualifies type  $T$  with a fresh qualifier variable if needed. The next line defines  $create(C)$ , which our type rules use to build the class table  $CT$ . The notation  $P \vdash L$  means program  $P$  contains the class definition  $L$  (the program  $P$  is an implicit global here). The function  $create$  makes an object type from a class by adding fresh qualifiers to field and method types, where needed. Note that we always create a fresh reference-level qualifier here for simplicity, and elide the detail of applying qualifier constants according to their level.

**Subtyping** The next step is to extend the subtype order among qualifiers to an ordering on types, as shown in Figure 4(c). (SUB-NULL) makes *null* a subtype of any other type, and (SUB-QTYPE) propagates subtyping from qualified types to qualifiers. Note that we ignore the base types, because we assume that the program is correct with respect to the standard types.

The third rule, (SUB-METHOD), propagates subtyping contravariantly to the domain and covariantly to the range. Subtyping on methods arises from overriding inherited methods, and thus if we assume the input program passes the Java type checker, then the types  $\tau_i$  and  $\tau'_i$  must in fact have equal base types, though they may differ in their qualifiers. (SUB-METHOD) treats the type of this contravariantly, since the receiver object is a method input. This constraint only makes sense because (SUB-QTYPE) does not check the base class types, which are covariant in method overriding.

**Class Table Construction** Figure 4(d) presents the first stage of type inference, constructing the class table  $CT$  by applying (CTCLASS) once to each class definition in the program. This rule uses  $create$  to build an object type for each class. This rule also constrains any methods in  $C$  to be subtypes of methods they override, using the auxiliary function  $mtype(m, C)$ , which looks up method  $m$  in class  $C$ . The subtyping constraint is ignored if  $mtype(m, D)$  does not exist. The function  $mtype$  is defined by (MTYPE-IN), which applies if  $m$  is a member of  $C$ , and (MTYPE-NIN), which applies otherwise and looks up the method in the parent class. The base class Object has no methods or fields.

In essence, by adding subtyping constraints between overriding and overridden methods, we are pre-computing a *class hierarchy analysis* [12], in which we assume that a call to a method  $m$  of class  $C$  might invoke any method that overrides  $m$  in a subclass. Though this approach is less precise than other methods, it fits well with a source-level type system extension, since a programmer would most likely expect that sub- and superclass qualifiers must be related.

We analogously define a function  $fype(f, \tau)$ , which looks up field  $f$  in  $C = class(\tau)$ . We pass  $fype$  a  $\tau$  instead of a  $C$  in anticipation of FS JQual (Section 3.2). As before, if  $f$  is a member of  $C$  then (FTYPE-IN) returns its type in  $CT(C)$ , and otherwise (FTYPE-NIN) returns its type from the parent of  $C$ .

**Type Inference Rules** Figure 4(e) shows the next stage of type inference, which traverses the method bodies in the program and generates subtyping constraints. The base judgment  $CT \vdash L$ , defined by (CLASS), infers types for the methods. Note that fields have no initializers in this language, and hence there is nothing to check for them. (METHOD) retrieves the type for  $m$  from  $CT$  and infers the type of  $e$ , assuming types for this and method parameters as given by  $CT$ . Notice that we need not re-translate the  $T$  types here, because class table construction has already done so. The last hypothesis of this rule constrains the type of  $e$  to be a subtype of the method return type.

The remaining rules prove judgments of the form  $CT, \Gamma \vdash e : \tau$ , meaning expression  $e$  has type  $\tau$  with class table  $CT$  and type environment  $\Gamma$ . (VAR), (NULL), and (SEQ) are standard, and (NEW) creates a freshly qualified type from  $T$ .

There are two type cast rules. (CAST-QTYPE) applies when the cast-to type contains a qualifier. This rule creates a freshly qualified type  $\tau'$  for the type of the cast. Notice that there is no relation between  $\tau$  and  $\tau'$  here—the cast “breaks the flow” of qualifiers. On the other hand, (CAST-TYPE) applies when the cast-to type is a bare type  $C$ . In this case, we make a constraint  $\tau \leq \tau'$  between the cast-from and the cast-to type. Recall that (SUB-QTYPE) ignores base types, and so this constraint in effect only relates the qualifiers on  $\tau$  and  $\tau'$ . We provide these two distinct type casts to give the programmer flexibility in specifying whether qualifiers propagate through casts.

(FREAD) infers the type  $\tau$  of the expression  $e$ , looks up field  $f$  in  $CT$ , and returns the type of its contents. (FWRITE) is similar, and it also constrains the type of  $e_2$  to be a subtype of the field contents type. Lastly, (INVOKE) looks up a method type using  $mtype$ , and then constrains the actual argument types to be subtypes of the formal arguments. We also require the type of the receiver object to be a subtype of this’s type in  $m$ .

**Constraint Resolution** We view the type rules in Figure 4(c)–(e) as *generating* the subtyping constraints they have in their hypotheses. These constraints have the forms  $\tau \leq \tau'$ ,  $\mu \leq \mu'$ , and  $Q \leq Q'$ . By applying the rules in

<ul style="list-style-type: none"> <li>• = <i>tracked</i></li> <li>○ = <i>not tracked</i></li> </ul> $\begin{aligned} \text{fresh}(C) &= (C^\bullet, \text{create}_\bullet(C)) && \kappa \text{ fresh} \\ \text{fresh}(C^Q) &= (C^Q, \text{create}_\bullet(C)) \end{aligned}$ $\begin{aligned} \text{create}_t(C) &= \{\text{create}_t(F_1); \dots; \text{create}_t(M_1); \dots\} \\ &\quad \text{where } P \vdash \text{class } C \text{ extends } D \{F_1; \dots; M_1; \dots\} \\ \text{create}_t(T \ f_{t'}) &= f : \text{ref}^\kappa(\text{fresh}(T)) && t = t', \kappa \text{ fresh} \\ \text{create}_t(T \ f_{t'}) &= \emptyset && t \neq t' \\ \text{create}_t(T_0 \ m(T_1 \ x_1, \dots, T_n \ x_n) \ T \ \{e\}) &= && t = \circ \\ &\quad (\text{fresh}(T_1) \times \dots \times \text{fresh}(T_n)) \ \text{fresh}(T) \rightarrow \text{fresh}(T_0) \\ \text{create}_t(T_0 \ m(T_1 \ x_1, \dots, T_n \ x_n) \ T \ \{e\}) &= \emptyset && t = \bullet \end{aligned}$ <p style="text-align: center;">(a) Translation from source to inference types</p>	<div style="display: flex; justify-content: space-around;"> <div style="text-align: center;"> <math display="block">\frac{Q \leq Q' \quad o \leq o'}{(C^Q, o) \leq (D^{Q'}, o')}</math> <p>(SUB-QTYPE)</p> </div> <div style="text-align: center;"> <math display="block">\frac{Q \leq Q' \quad \tau \leq \tau' \quad \tau' \leq \tau}{\text{ref}^Q(\tau) \leq \text{ref}^{Q'}(\tau')}</math> <p>(SUB-FIELD)</p> </div> </div> <div style="text-align: center; margin-top: 10px;"> <math display="block">\frac{\phi_i \leq \phi'_i \quad i \in 1..n}{\{f_1 : \phi_1; \dots; f_n : \phi_n\} \leq \{f_1 : \phi'_1; \dots; f_n : \phi'_n\}}</math> <p>(SUB-OTYPE)</p> </div> <p style="text-align: right;">(b) Subtyping rules</p>
<div style="display: flex; justify-content: space-around; margin-bottom: 10px;"> <div style="text-align: center;"> <math display="block">\frac{\text{CTCLASS} \quad \text{CT}(C) = \text{create}_\circ(C) \quad \text{CT} \vdash \text{mtype}(m, C) \leq \text{mtype}(m, D) \quad \forall m \in C}{\text{CT} \vdash C \text{ extends } D \{ \dots; M_1; \dots; M_n \}}</math> </div> <div style="text-align: center;"> <math display="block">\frac{\text{FTYPE-TRACK} \quad \tau = (C^Q, o) \quad o = \{ \dots; f : \phi; \dots \}}{\text{CT} \vdash \text{ftype}(f, \tau) = \phi}</math> </div> </div> <div style="display: flex; justify-content: space-around;"> <div style="text-align: center;"> <math display="block">\frac{\text{FTYPE-IN} \quad \tau = (C^Q, o) \quad P \vdash C \text{ extends } D \{F_1; \dots; F_n; \dots\} \quad f \notin \text{dom}(o) \quad F_i = T \ f_o}{\text{CT} \vdash \text{ftype}(f, \tau) = \text{CT}(C)(f)}</math> </div> <div style="text-align: center;"> <math display="block">\frac{\text{FTYPE-NIN} \quad \tau = (C^Q, o) \quad P \vdash C \text{ extends } D \{F_1; \dots; F_n; \dots\} \quad f \notin \text{dom}(o) \quad \forall i. F_i \neq T \ f_o}{\text{CT} \vdash \text{ftype}(f, \tau) = \text{ftype}(f, (D^Q, o))}</math> </div> </div> <p style="text-align: center;">(c) Class table construction and field lookup</p>	

**Figure 5.** Qualifier inference — FS JQual (modifications only)

Figure 4(c), we can eliminate the first two kinds of constraints, so that we are left with a set of qualifier constraints  $Q \leq Q'$ . Assuming that the partial order on qualifier constants is a lattice, we can solve these constraints using graph reachability to look for inconsistent paths through the graph [15]. For example, a path from `readonly` to `mutable` would be inconsistent, since it corresponds to a constraint `readonly`  $\leq$  `mutable`. If the constraints have a solution, then we have found a valid typing, and if not, then we report a type qualifier error. Section 3.3, below, gives several examples of constraint graphs.

### 3.2 Field-Sensitivity

As mentioned in the introduction, one important tradeoff when performing a static analysis is between *field-based* and *field-sensitive* analysis. To understand the difference, consider the following code snippet:

```

1      class C extends Object { int f; }
2      C c1 = new C;
3      C c2 = new C;
4      opaque int x = c1.f;

```

Line 1 defines a class `C` with a field `f`. Then lines 2 and 3 create two instances of `C`, and line 4 stores `c1.f` in an opaque integer. Therefore `c1.f` must itself be opaque, since transparent integers may not be passed to opaque positions. In Core JQual, all instances of `C` share the same field types, no

matter how often `C` is instantiated. Thus `c2.f` would also be considered opaque in this example. On the other hand, in a field-sensitive analysis, each syntactic occurrence of a class is given its own set of field types. Thus we would infer that `c1.f` is opaque, but `c2.f` could be transparent, since the type of `c2.f` is unconstrained.

Clearly a field-sensitive analysis can be much more precise than a field-based analysis, but field-sensitivity might not make sense for all applications—after all, in standard Java classes, there are many fields with non-generic types. Additionally, the more polymorphism there is in a type, the harder it may be to understand, especially when polymorphism is combined with subtyping. Field-based analysis can also be more efficient than field-sensitive analysis, since there is less information to track about the program. Indeed, our current implementation of JQual (Section 4) typically runs out of memory when trying to perform full field-sensitive analysis on our larger benchmark programs, and efficient field-sensitive analysis is an active area of research [27, 31, 43, 44, 48].

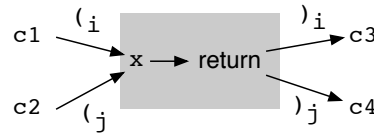
JQual solves this problem by asking the programmer to mark fields that should be treated field-sensitively. We call such fields *tracked*. In our experience, we found that it is usually easy to determine what fields to make tracked, and that only a few tracked fields are needed.

We designed Core JQual carefully so that FS JQual, our field-sensitive inference system, is a minimal extension to it.

```

1  class D extends Object {
2      C id(C x) {
3          return x;
4      }
5      void main() {
6          C c1 = new C(); C c2 = new C();
7          C c3 = id(c1); C c4 = id(c2);
8      }
9  }

```



**Figure 6.** Context-sensitivity example and constraint graph

Formally, field declarations in the source code are now of the form  $T f_t$ , where  $t$  is either  $\bullet$  for a tracked field, or  $\circ$  for a non-tracked field. We extend types  $\tau$  to the form

$$\tau ::= null \mid (C^Q, o)$$

Here the  $C^Q$  component is as before, and the object type  $o$  maintains tracked fields. Non-tracked fields are stored in the class table. For example, suppose  $C$  is declared as

```
class C extends D { T f $\bullet$ ; T' g $\circ$  }
```

Then  $CT$  maps class  $C$  to a type  $\{g : \phi'\}$ , and each occurrence of  $C$  in the program text has a type  $\tau = (C^Q, \{f : \phi\})$ . During inference, when we look up  $f$  in type  $\tau$ , we return  $\phi$ , while when we look up  $g$  in type  $\tau$ , we find its type in  $CT$  and return  $\phi'$ —and thus in our example, each separate occurrence of  $C$  will have its own type for field  $f$ , but they will all share the type of field  $g$ . We define  $class((C^Q, o)) = C$ .

Figure 5 shows the necessary changes to the typing rules. Part (a) of this figure redefines *fresh* to return a  $\tau$  where the object type portion of the  $\tau$  is constructed with a call to  $create_\bullet$ . The *create* function now has a subscript  $t$  indicating whether the object type it returns should include tracked or non-tracked fields. In particular, when  $create_\bullet(T f_{t'})$  is called, it generates a field type for  $f$  if  $t'$  is tracked, and otherwise it does not, denoted by returning  $\emptyset$ . Calling  $create_\circ$  does the opposite. Methods are shared across all instances, and thus they reside only in  $CT$  and are generated with  $create_\circ$ , and  $create_\bullet$  never creates method types.

Notice that if  $create_t$  is applied to a recursive type with a tracked field, we could potential enter an infinite loop, unrolling the type indefinitely. To prevent this, when we encounter an occurrence of class  $C$  nested at any depth within class  $C$ , we force them to share the same object type (not shown in the figure).

Figure 5(b) shows the necessary changes to the subtyping rules. (SUB-QTYPE) is modified to propagate subtyping to object types. (SUB-FIELD) handles subtyping on fields. Since *ref* is a nonvariant constructor, we require that  $\tau$  and  $\tau'$  are equal. Lastly, (SUB-OTYPE) propagates subtyping from an object type to its components covariantly. Note that we only apply this rule to object types representing tracked fields, and thus the object types do not contain methods. Also notice that this rule requires that the sub- and superclass have the same fields and methods, i.e., it does not allow width sub-

typing. We need this feature in practice to propagate qualifiers on tracked fields through up- and down-casts. For example, consider the code

```
Foo f = ...; Object x = f; Foo g = (Foo) x;
```

If  $Foo$  contains a tracked field, we must add it to  $x$ 's type so that its qualifiers reach  $g$ .

Finally, Figure 5(c) gives the new rules for class table construction and field lookup. (CTCLASS) is the same as before, but we use  $create_\circ$  to make object types with non-tracked fields. When applying  $f_{type}(f, \tau)$  to look up a field type, there are three cases. (FTYPE-TRACK) applies when  $f$  is part of the object type stored in  $\tau$ , i.e., when  $f$  is tracked. Otherwise, (FTYPE-IN) and (FTYPE-NIN) retrieve the field from the class table as before.

The class, method, and expression typing rules from Core JQual are unchanged in FS JQual. The modifications to the field lookup rules do essentially all the necessary work. For example, (FRREAD) still invokes  $f_{type}(f, \tau)$  to compute a type for  $e.f$ , but it uses the new field lookup rules in Figure 5. Similarly, constraints such as  $\tau'_i \leq \tau_i$  generated in (INVOKE) now operate on types  $(C^Q, o)$ , using the additional subtyping rules. Constraint resolution is also the same process as in Core JQual: We generate subtyping constraints, reduce them to qualifier constraint  $Q \leq Q'$ , and solve the constraints using graph reachability.

### 3.3 Context-Sensitivity

Along with field-sensitivity, another major tradeoff in a static analysis is whether to use context-sensitivity (a.k.a. parametric polymorphism for type systems [32]). Consider the code example shown on the left in Figure 6. Line 1 defines a class  $D$ , and lines 2–4 define an identity method  $id$  on some other class  $C$ . On lines 6–7 we create two instances,  $c1$  and  $c2$ , and pass them through  $id$ , storing the results in  $c3$  and  $c4$ , respectively. Since Core and FS JQual are context-insensitive, they would determine that the qualifiers on  $c1$  and  $c2$  flow to (i.e., are subtypes of) the qualifiers on *both*  $c3$  and  $c4$ . This is imprecise, because at run time,  $c1$  only flows to  $c3$ , and  $c2$  only flows to  $c4$ . We can solve this problem by introducing context-sensitivity, so that we can distinguish the two calls to  $id$ .

JQual incorporates context-sensitivity using a technique proposed by Reps et al [38] and Rehof et al [36]. In this

<div style="display: flex; justify-content: space-between;"> <div style="width: 45%;"> <p>(INST-FIELD)</p> <math display="block">\frac{Q \preceq_{\rho}^j Q' \quad \tau \preceq_{\rho}^j \tau' \quad \tau \preceq_{-\rho}^j \tau'}{ref^Q(\tau) \preceq_{\rho}^j ref^{Q'}(\tau')}</math> </div> <div style="width: 45%;"> <p>(INST-QTYPE)</p> <math display="block">\frac{Q \preceq_{\rho}^j Q' \quad o \preceq_{\rho}^j o'}{(C^Q, o) \preceq_{\rho}^j (D^{Q'}, o')}</math> </div> </div> <p>(INST-OTYPE)</p> $\frac{\phi_i \preceq_{\rho}^j \phi'_i \quad i \in 1..n}{\{f_1 : \phi_1; \dots; f_n : \phi_n\} \preceq_{\rho}^j \{f_1 : \phi'_1; \dots; f_n : \phi'_n\}}$ <p style="text-align: center;">(a) Instantiation rules</p>	<p>(FTYPE-IN)</p> $\frac{\tau = (C^Q, o) \quad P \vdash C \text{ extends } D \{F_1; \dots; F_n; \dots\} \quad f \notin \text{dom}(o) \quad F_i = T f_o}{\phi = CT(C)(f) \quad \phi \preceq_{+}^* \phi \quad \phi \preceq_{-}^* \phi}$ <p style="text-align: center;">(b) Field lookup</p>
<p>(INVOKE)</p> $\frac{CT, \Gamma \vdash e : \tau_e \quad CT \vdash \text{mtype}(m, \text{class}(\tau_e)) = (\tau_1 \times \dots \times \tau_n) \tau \rightarrow \tau_0 \quad \tau_0 \preceq_{+}^j \tau_0' \quad CT, \Gamma \vdash e_i : \tau'_i \quad \tau_i \preceq_{-}^j \tau'_i \quad i \in 1..n \quad \tau \preceq_{-}^j \tau_e}{CT, \Gamma \vdash e.m^j(e_1, \dots, e_n) : \tau_0'}$ <p style="text-align: center;">(c) Invoke expression typing</p>	

**Figure 7.** Qualifier inference — CS/FS JQual (modifications only)

approach, we reduce context-sensitive inference to the problem of *context-free language (CFL) reachability* on a constraint graph. As is well-known, context-sensitive analysis can be more expensive than context-insensitive analysis, but CFL reachability has proved to be very scalable in practice [15, 36, 43, 44].

We can think of a regular qualifier constraint  $Q \leq Q'$  as an edge  $Q \rightarrow Q'$  in a graph, where the nodes in the graph are qualifier constants and variables. As mentioned earlier, we solve a set of qualifier constraints by looking for inconsistent paths in the constraint graph. We add context-sensitivity by introducing edges labeled with indexed parentheses. We pick a fresh index  $j$  for each call site in the program. Then instead of regular edges, when qualifiers are passed into a method at call site  $j$ , we add *instantiation edges* labeled with  $(_j$ . Similarly, when qualifiers are returned from the same call, we use instantiation edges labeled with  $)_j$ . To check satisfiability, we use context-free language reachability to propagate qualifiers only along paths that have no mismatched parentheses.

For example, the right part of Figure 6 shows the key portion of the constraint graph for the sample program. Here the first call to `id` is indexed by  $i$ , and the second call is indexed by  $j$ . Notice that the path from `c1` to `c3` is valid, because  $(_i$  matches  $)_i$ , and similarly for the path from `c2` to `c4`. However, the path from `c1` to `c4` is invalid, because  $(_i$  does not match  $)_j$ , and similarly for the path from `c2` to `c3`. Thus we have gained precision by excluding *unrealizable paths* from our analysis, which correspond to non-matched call and return sequences [38]. Notice that these paths would have been valid under a monomorphic analysis, which would produce the same graph but with no edge labels.

Figure 7 shows the changes necessary for CS/FS JQual, which extends FS JQual with context-sensitivity using the approach just outlined. Formally, labeled edges in the constraint graph correspond to *instantiation constraints* of the form  $Q \preceq_{\rho}^j Q'$ . Here  $j$  indicates the call site, and  $\rho$  is the *variance*, either  $-$  for contravariant positions or  $+$  for covariant positions. The constraint  $Q \preceq_{+}^j Q'$  corresponds to an edge  $Q \rightarrow^j Q'$ , and the constraint  $Q \preceq_{-}^j Q'$  corresponds to an edge  $Q' \rightarrow^j Q$  (notice that we flip the direction of the arrow for this last constraint [36]).

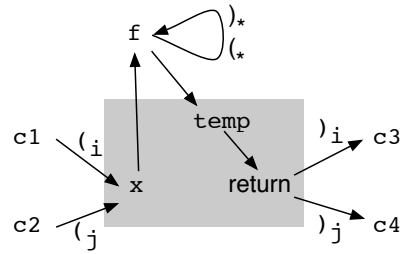
Part (a) of the figure extends instantiation constraints to types. (INST-FIELD) propagates an instantiation constraint from a reference type to its components. Here  $-\rho$  is the opposite of the variance  $\rho$ , and notice that, just as with subtyping, references are both co- and contravariant in their contents types. (INST-QTYPE) and (INST-OTYPE) both propagate an instantiation constraint covariantly to the components of the types. Rather than give a separate rule for *null*, we assume we always apply subtyping to turn *null* types into non-*null* types before using them in instantiation constraints.

The field lookup rules for CS/FS JQual are the same as FS JQual, except (FTYPE-IN), as shown in Figure 7(b). This rule applies to non-tracked fields, which are shared across all instances, and thus must be considered global by the analysis. To allow propagation of qualifiers through these fields from any call site to any call site, (FTYPE-IN) adds the constraints  $\phi \preceq_{\rho}^* \phi$  for  $\rho = +$  and  $\rho = -$ . The label  $*$  is a special index that matches any call site in the program, and thus these constraints, which correspond to self-loops in the constraint graph, allow the correct flow for globals [36]. In our implementation, we also add these constraints for static fields, since they are global as well. We illustrate self-loops in an example shortly.

```

1      class D extends Object {
2          static C f;
3          C foo(C x) {
4              C temp = this.f;
5              this.f = x;
6              return temp;
7          }
8          void main() {
9              C c1 = new C(); C c2 = new C();
10             C c3 = foo(c1); C c4 = foo(c2);
11         }
12     }

```



**Figure 8.** Self-loops in constraint graphs

Figure 7(c) shows the one change necessary to the expression type rules. The revised (INVOKE) rule creates instantiation edges rather than subtyping constraints at the call site. Argument types and the receiver object type are contravariant, and the return type is covariant. For example, consider Figure 6 again. At the first call to `id`, this rule generates two constraints (ignoring the receiver):  $x \preceq_-^i c1$ , corresponding to the edge from `c1` to `x`, and the constraint  $\text{return} \preceq_+^i c2$ , corresponding to the edge from `return` to `c2`. The edges for the second call to `id` are similar.

Notice that (INVOKE) generates instantiation constraints on the method based on the compile-time type of  $e$ . By (CTCLASS) from Figure 4, we have already added appropriate constraints between this method type and the types of methods related to it by overriding in the class hierarchy. Thus in essence, (INVOKE) instantiates the type of a method that represents all possible run-time objects represented by  $e$ .

To understand the need for self-loops in (FTYPE-IN), consider the example in Figure 8. The method `foo` on lines 3–7 stores the value of its argument `x` in a static field `f`, and then returns the old value of `f`. Then on lines 9–10, we create new instances `c1` and `c2` and pass them through `foo`, storing the results in `c3` and `c4`, respectively. Notice that when this code runs, the value of `c1` will be stored in `c4`.

The right part of the figure shows the constraints generated for this example. Since `f` is a static field, we treat it like a non-tracked field and add self-loops labeled with  $(*)$  and  $(*)$ . Thus there is a matched path from `c1` to `c4` that crucially uses the self-loops on `D.f`. Notice that without these self-loops, we would miss this path in the graph and be unsound.

While context-sensitivity does increase the complexity of the inference algorithm (up to cubic time [36]), our experiments (Section 4.2) show that it is scalable in practice and increases precision.

## 4. Implementation and Experiments

We implemented JQual as an Eclipse plug-in. JQual performs its analysis on source code, using Eclipse’s Java Development Toolkit for parsing. JQual generates and solves constraints using CQual’s back-end, and if any constraints

are unsatisfiable, meaning that some value in the program has inconsistent inferred qualifiers, then the constraint solver issues a warning message that includes an inference path illustrating the error [15]. CQual uses heuristics to suppress excess warnings, but we found that they did not always work, and so the counts of warning messages in our experiments do not include warnings with duplicate paths.

The input to JQual is a set of source files to analyze and a configuration file describing the order among the qualifier constants. Type qualifier annotations are given in source code as Javadoc comments with custom tags. We allow methods, fields, variables, and type casts to have qualifier annotations, and fields may also be annotated as tracked. We chose Javadoc instead of Java 1.5-style annotations [6] because the latter cannot appear in every position that we needed for our experiments. (JSR 308 proposes a solution to this problem [13].) Qualifier annotations also indicate whether to apply the qualifier to the reference or value level of a variable.

When the user runs JQual, they can choose whether to enable field-sensitivity, context-sensitivity, both, or neither. In our formalism, the function  $\text{create}_\bullet$  always creates fresh qualified types for each tracked field, but this would be prohibitively expensive in practice. Instead, JQual creates fields lazily, on-demand. Initially,  $\text{create}_\bullet$  returns a type with an empty set of fields. Then when  $\text{ftype}(f, \tau)$  is called, we add  $f$  to  $\tau$  if it is a possible tracked field of the object. As a heuristic, when we generate a regular subtyping constraint  $(C^Q, o) \leq (D^{Q'}, o')$ , we unify  $o$  and  $o'$ , so that they share tracked field sets. This results in little lost precision because fields are reference types, and so in any case the fields’ value-level qualifiers would be equal after such a constraint. We do not unify field sets across instantiation constraints.

JQual handles a number of features of Java not included in our formal system. Constructors are modeled similarly to ordinary methods, except the return type is the same as the newly constructed object type. JQual is a whole-program analysis, and so it requires source code for all necessary classes. JQual begins from the initial set of source files, transitively finds all classes they reference, and analyzes them. JQual aims to be sound, but has three potential sources of

unsoundness: Implicit `super()` constructor calls are ignored, we do not track the flow of qualifiers from `throw` to `catch` clauses, and we do not model native method or reflective API calls specially, and so they can lose flow of qualifiers. We do not feel that any of these are significant limitations.

We performed all of our experiments on an 2.4GHz AMD Athlon 4600 processor with 4GB of memory.

#### 4.1 Qualifiers for the JNI

For our first experiment with JQual, we checked the JNI qualifiers described in Section 2.1 on a small benchmark suite. Rather than add qualifier annotations manually, we developed a simple tool to do so automatically. Our tool uses CIL [33] to analyze C code and find places where parameters and return values of native methods are directly cast from an `int` to some C type `T` or vice-versa. If `T` is a pointer type, then we mark the corresponding `int` as `opaque`, and if `T` is an enumeration with maximum value  $i$ , we mark it as `enumi`. Any other parameter or return types are not qualified. Note that our analysis may omit some qualifier annotations due to its handling of certain C constructs, and because it ignores any transitive flow of Java ints through the C code.

After performing this first step, we then add the qualifiers to the Java code. Rather than modify the actual Java source, we used a separate auxiliary file to specify qualifiers. We also extended JQual so that any integers manipulated by Java, such as numeric literals or the results of arithmetic operations, are transparent. Additionally, several of the benchmarks we analyzed form a library that is intended to be used in many different applications. Without knowing exactly what the client code is, we wanted to check whether any transparent values could be used in opaque positions. Thus we modified JQual so that any ints that could come from outside the scope of the library package, namely public method arguments and public fields, are transparent. Finally, in this analysis, we do not model the Java standard library, and we do not generate any constraints when referring to unavailable code. These choices introduce some potential unsoundness, but keep the experiment simple.

Figure 9 summarizes our benchmark suite and the results of running JQual. The first set of benchmarks, listed above the line, is a collection of related libraries for accessing the Gimp Toolkit (GTK). We analyzed each of these in isolation and then together, listed on the top line. The other two benchmarks in the chart, listed below the line, are separate programs. The second column of the table lists the number of lines of code of each benchmark. The third column lists the number of qualifiers added to native methods based on our C code analysis, and the fourth column counts the total number of parameter and return positions on native methods. Over all the benchmarks, roughly 28% of the possible positions were annotated with qualifiers, indicating that passing C pointers and enumerations between Java and C is relatively common in the JNI.

To apply JQual to these benchmarks, we used the following procedure. We began by running JQual context- and field-insensitively. We found that this generated a very large number of warnings, and so we immediately enabled context-sensitivity. The resulting warning counts, still field-insensitive, are shown in the fifth column in Figure 9. We felt that this was still a large number of warnings, and so we inspected the output to identify fields that seemed like good candidates for field-sensitivity, which we then marked as tracked. We continued re-running the analysis and looking for tracked fields until we could not identify any more good candidates. The results of the final context- and field-sensitive experiment are shown in the last three columns of Figure 9. The first column lists the running time in seconds (one run) the next column lists the total number of tracked fields, and the last column lists the number of warnings reported by JQual.

As these results show, most of the opaque and enum-qualified integers are used safely in the Java code, since there are few warnings overall. We inspected the final warnings for `combined.gtk` and the two standalone benchmarks manually, and found they fell into four categories. We did not find any outright errors, but 7 of the warnings occur because integers arguments of public methods could flow to opaque arguments of native methods. This is a bad programming practice because it allows library clients to pass transparent integers to opaque positions. The remaining warnings are all false positives. Seven of the warnings occur when the integer literal 0, which our analysis qualifies as `enum0`, is passed to an opaque position, meaning it is used as the C null pointer. Two of the warnings are from our assumption about enumeration types in C. In this case, rather than representing a range of values, the enumerations represented bit flags, and our analysis marked the results of bit operations as transparent, even though they formed legitimate values. Another 23 warnings were related to our analysis of enumeration types, but involved inference paths that appear to be unrealizable. The last warning was due to an extra cast inserted by CIL that caused our C code analysis to qualify to an integer that is not a pointer or enumeration.

#### 4.2 Immutability Inference

For our second experiment with JQual, we used JQual to infer readonly qualifiers for a selection of Java programs. To add these qualifiers to JQual, we modified two typing rules as shown in Figure 10. In (FREAD), we generate a new constraint  $Q \leq Q_f$ , where  $Q_f$  is the qualifier on the type of  $e.f$  and  $Q$  is the qualifier on  $e$ . In this way, if  $Q$  is readonly, then  $Q_f$  will be as well, i.e., if  $e$  is readonly, then so is  $e.f$ . In (FWRITE), we generate a new constraint  $Q \leq \text{mutable}$ , since field  $f$  is written to. Here  $Q$  is  $f$ 's reference-level qualifier. We also generate a constraint  $Q' \leq \text{nonfinal}$ , where  $Q'$  is the value-level qualifier on field  $f$ . Here we can clearly see that `nonfinal` is a reference level qualifier and `mutable` is value level qualifier. As mentioned in Section 2.2 we allow

Benchmark	KLoC	Qual	Pos	Fld Ins	Fld Sens		
				Wrn	Time	Trck	Wrn
combined_gtk	40.6	4,929	17,562	187	2:18s	22	37
libgtk-java-2.6.2	32.4	4,029	13,844	151	2:07s	20	11
libvte-java-0.11.11	0.2	47	158	0	0:00s	0	0
libglade-java-2.10.1	1.0	5	20	0	0:03s	0	0
libgconf-java-2.10.1	0.7	148	504	0	0:00s	0	0
libgnome-java-2.10.1	5.1	632	2,652	3	0:06s	0	0
libgtkmozembed-java-1.7.0	0.5	18	80	0	0:00s	0	0
libgtkhtml-java-2.6.0	0.7	50	304	1	0:01s	2	1
jnetfilter	1.2	77	432	24	0:04s	6	3
libreadline-java-0.8.0	0.3	2	36	0	0:00s	0	0

Figure 9. Results of JNI Experiments

$$\begin{array}{l}
\text{(FREAD)} \\
\frac{CT, \Gamma \vdash e : \tau \quad \tau = (C^Q, o) \quad CT \vdash \text{ftype}(f, \tau) = \text{ref}^{Q'}(\tau') \quad \tau' = (D^{Q_f}, o') \quad Q \leq Q_f}{CT, \Gamma \vdash e.f : \tau'} \\
\text{(FWRITE)} \\
\frac{CT, \Gamma \vdash e_1 : \tau \quad \tau = (C^Q, o) \quad CT \vdash \text{ftype}(f, \tau) = \text{ref}^{Q'}(\tau') \quad CT, \Gamma \vdash e_2 : \tau'' \quad \tau'' \leq \tau' \quad Q \leq \text{mutable} \quad Q' \leq \text{nonfinal}}{CT, \Gamma \vdash e_1.f = e_2 : \tau''}
\end{array}$$

Figure 10. Modified inference rules for immutability

constructors to write to readonly and final fields, and so when analyzing constructors we use the original versions of (FREAD) and (FWRITE) rather than the modified versions.

We applied immutability inference to a variety of Java programs. Figure 11 summarizes the results. The programs beginning with underscores are part of the SPEC JVM benchmark suite [2]. The others are open source programs downloaded from SourceForge [1]. For the SPEC benchmarks, we included the code of the SPEC JVM execution framework in our analysis, and thus the line counts (KLoC) are larger than a straight count of the benchmarks' code. To model library calls, we created a special stub version of the libraries that included mutable and nonfinal annotations. We also annotated fields of container classes as tracked, since we expect containers to be used polymorphically.

We ran JQual with three of the four possible combinations of context-insensitive (CI) versus context-sensitive (CS) and field-based (FB) versus field-sensitive (FS). We omitted CI/FS analysis, since context-insensitive constructors would merge the fields of different instances, reducing or eliminating the benefit for field-sensitivity. For the various configurations, we report the number of readonly (RO) positions inferred on object types in method signatures (including arguments, return, and this); then the number of fields inferred to be final (FF); and then the running time (one run) We do not include methods and fields from the library stubs in our counts.

In the context-sensitive analyses, we counted a method parameter or result as readonly if it could be used polymor-

phically as either mutable or readonly. For the field-based analysis, we counted each field from a class once, since field types are shared across all instances of a class. In the field-sensitive analysis, we counted each instance of a tracked field separately. We omit the FF column from the CS/FB experiment because context-sensitivity does not change where we may infer final.

The average percentage of readonly method positions ranges from 48% to 62%, suggesting that large number of parameters, return values, and receiver objects in methods are readonly. A smaller, but still significant percentage of fields are inferred to be final. As expected, the precision increases with the addition of context- and field-sensitivity. In most cases, increased precision comes at the cost of increased running time, though in a few cases the running time actually decreases, most likely because there are fewer valid paths for propagating qualifiers.

To better understand the results of this experiment, we selected fifty method signature positions, determined whether they were readonly or mutable according to the analysis, and then manually inspected the code. We found that the qualifiers inferred on 35 positions we looked at were non-trivial and seemed quite useful, describing accurate properties of the code. Another 3 positions were inferred readonly but were Strings, which are clearly immutable, and 5 more were return values of methods that are never called—hence the return value is trivially readonly. Lastly, 7 of the positions in method signatures were mutable, but seemed likely to be readonly if we had made more fields tracked.

Overall, our results show that JQual is able to discover many useful cases of immutability across our benchmarks.

## 5. Related Work

There are several threads of work related to JQual. In our own prior work, we proposed type qualifiers as a general mechanism for lightweight static checking and described CQual [15], which adds type qualifiers to C. Among other applications, CQual has been used to infer const qualifiers [14], to find format-string vulnerabilities [42], and to find user-kernel pointer vulnerabilities [24] and deadlocks [4, 16] in the Linux kernel. Several other researchers have also used CQual [9, 18, 49], and our hope is that JQual

Benchmark	KLoC	CI/FF			CS/FF		CS/FS		
		RO (%)	FF (%)	Time	RO (%)	Time	RO (%)	FF (%)	Time
jdbm	4.2	283 (33%)	182 (41%)	0:02s	379 (44%)	0:01s	402 (46%)	301 (52%)	0:02s
_227_mtrt	5.7	358 (56%)	160 (22%)	0:02s	398 (62%)	0:02s	446 (70%)	261 (30%)	0:03s
_201_compress	6.2	368 (53%)	190 (23%)	0:03s	418 (60%)	0:02s	466 (67%)	325 (32%)	0:03s
_209_db	6.4	364 (53%)	162 (21%)	0:02s	408 (60%)	0:02s	456 (67%)	276 (31%)	0:03s
_200_check	7.0	481 (62%)	206 (26%)	0:03s	527 (68%)	0:02s	576 (74%)	310 (33%)	0:04s
_205_raytrace	7.7	521 (53%)	190 (21%)	0:03s	602 (61%)	0:02s	650 (66%)	325 (30%)	0:04s
_202_jess	12.0	412 (57%)	164 (22%)	0:06s	460 (64%)	0:03s	508 (71%)	287 (31%)	0:05s
jgap	10.2	784 (41%)	460 (44%)	0:07s	1,003 (52%)	0:04s	1,087 (57%)	656 (53%)	0:06s
jgraph	11.9	1,167 (41%)	288 (29%)	0:10s	1,453 (51%)	0:06s	1,453 (51%)	3,828 (71%)	0:08s
jtds	21.5	1,040 (35%)	854 (48%)	0:10s	1,247 (42%)	0:05s	1,323 (44%)	927 (49%)	0:08s
_213_javac	45.7	359 (56%)	160 (22%)	0:02s	399 (63%)	0:02s	448 (70%)	261 (30%)	0:03s
jfreechart	121.4	9,377 (46%)	2,098 (33%)	0:55s	11,190 (55%)	0:56s	11,444 (57%)	5,573 (55%)	1:12s
<b>Average</b>		<b>(48%)</b>	<b>(29%)</b>	—	<b>(56%)</b>	—	<b>(62%)</b>	<b>(42%)</b>	—

Figure 11. Immutability inference results

will similarly become a platform for experimentation with lightweight static analysis of Java.

There are several challenges in performing static analysis of Java code as compared to C. We encountered the same issues that have been previously identified by other researchers [40, 26]. In Java, method invocations are almost all via dynamic dispatch, as opposed to C, where function pointer calls occur regularly but much less often. Thus we need to model the call graph in some way, and JQual’s choice is class-hierarchy analysis, to match Java’s type system. Another difference is that Java does not have pointers to the stack or pointer arithmetic, though it does have many dynamic allocation sites. The Java standard libraries are quite large, and analyzing them requires significant resources. In our experiments, we used annotated stubbed versions of required library classes. Lastly, reflection [30], dynamic class loading, and native methods make it difficult to achieve soundness in a static analysis of Java. These remain open problems in the research community and for JQual.

Type qualifier inference is closely related to the problem of points-to analysis, which has been an active area of research in recent years. The goal of points-to analysis for Java is to determine how run-time objects flow through the program. Similarly, in type qualifier inference, our goal is to determine how qualifiers flow through the program, and then additionally to check that the flow is valid with respect to the programmer-supplied qualifier ordering. Thus we may be able to use others’ techniques for points-to analysis to perform type qualifier inference. However, it is unclear whether arbitrary points-to analyses can support the extra conditions for readonly, in which reads through readonly references produce new readonly references, and whether they include analysis of the flow of integers, which we need for opaque and enum inference. In general, type qualifiers are intended as a lightweight, source-level specification and checking system, whereas points-to analysis is a core static analysis that is not directly reported to the programmer.

As discussed in Section 2.2, our readonly and mutable qualifiers are based on Javari [8, 47]. Tschantz [46] presents

an inference algorithm for Javari qualifiers. Tschantz’s algorithm includes some specialized notions of field-sensitivity, expressed by inferring bounds on instantiated type variables, as well as context-sensitivity, expressed with a *romaybe* qualifier. It is unclear exactly how these relate to the general notions of field- and context-sensitivity in JQual. Additionally, Tschantz reports only one result of his inference algorithm and suggests the implementation is incomplete.

Pratikakis et al [34] present a framework for programming with proxies in Java. Their system uses a proxy qualifier to mark proxied objects, and qualifier inference determines where these objects are used and hence must be demanded at run time. Their qualifier system is specialized to handle proxy, while JQual is general-purpose. An interesting future direction is using JQual to infer proxy qualifiers.

Chin et al [10] propose *semantic type qualifiers*, in which programmers specify user-defined type qualifiers for C using a type refinement rule language. In this framework, the refinement rules are automatically incorporated into the source language type checker and proven sound with respect to the qualifier semantics. Later work adds monomorphic type qualifier and refinement rule inference [11]. In contrast, JQual uses a fixed set of qualifier rules that can only be modified by editing JQual’s source code (e.g., the tweaks necessary for readonly and opaque checking), but JQual applies to Java and includes context- and field-sensitivity. Andreae et al [5] bring semantic type qualifiers to Java, but do not investigate qualifier inference.

Several researchers [27, 31, 39, 41] have developed analyses for discovering whether Java methods have side effects. This is similar to inferring readonly, but these systems are designed mostly for compiler optimization, rather than source-level specification. Liu and Milanova propose immutability inference for fields in Java [29]. This is different than the reference immutability for arguments and results inferred by JQual. Artzi et al [7] propose a combined static and dynamic mutability analysis for Java, using a different notion of parameter mutability than readonly.

Type qualifiers are related to the refinement types of Freeman and Pfenning [19]. While similar in spirit, refinement types are significantly more complex, based on the theory of intersection types, while qualifiers include only atomic subtyping. Qualifiers cannot express as rich a set of properties but may provide a simpler programming interface and allow a more efficient implementation. The work of Strom and Yemini [45] on tpestate verification similarly provides static checking of properties orthogonal to standard types, although the emphasis is on flow-sensitive properties.

We briefly discuss some approaches to points-to analysis for Java. Milanova et al [31] present an object-sensitive alias analysis for Java, in which each set of calls to a method with a different receiver class type is analyzed separately. They show that this limited form of context-sensitivity enhances precision while remaining efficient. JQual, in contrast, uses full context-sensitivity.

Sridharan et al [43, 44] express points-to analysis for Java as a context-free language (CFL) reachability problem. In their encoding, they use CFL reachability both for polymorphic method calls and for field-sensitivity. Since this leads to a potentially undecidable problem, to remain tractable they use an approximation and refinement scheme when solving the constraint graph. In contrast, JQual places fields structurally inside of types, and does not use CFL edges for field access. JQual's approach is much simpler technically, but in our experience does not scale if we enable full field-sensitivity. However, because we only enable field-sensitivity for individual fields, we avoid the scaling problem while still being precise enough for our applications.

Lhotak [27] describes a BDD-based framework for developing various static analyses of Java, including points-to analysis. Whaley and Lam [48] show how to use BDDs to compute a context-sensitive, field-sensitive alias analysis of Java that scales to large programs. Lam et al [25] generalize this approach to a number of different static analyses of Java. Both of these systems provide a generic infrastructure for program analysis, and may be useful for type qualifiers.

Reps has shown that using CFL reachability for both context- and field-sensitivity is undecidable [37]. We avoid this problem in CS/FS JQual by only using CFL reachability for context-sensitivity. We achieve field-sensitivity by represented tracked fields in the structure of types, and we “tie the knot” for recursive types whenever we encounter them. This last part, which is an approximation, maintains decidability.

Finally, our qualifiers for the JNI can be considered a follow on to prior work in which we performed type safety checking across the JNI [20]. Our prior work checked that C code used Java types safely, whereas in this paper we use JQual to check that Java code uses C types safely.

## 6. Conclusion

We have presented JQual, a system for adding type qualifiers and type qualifier inference to Java. We formalized Core

JQual, a field-based, context-insensitive type qualifier inference system for a variant of Featherweight Java. We then presented FS JQual, a small extension to this system that adds field-sensitivity, and CS/FS JQual, which further adds context-sensitive inference using CFL reachability. We studied two major applications of JQual: checking that pointer and enum values passed through a JNI API are used correctly in a Java program, and inferring readonly and final qualifiers in Java source code. JQual found several examples of potential opaque violations in a small benchmark suite, and was able to infer that many positions are readonly or final. These results suggest that user-defined type qualifiers can provide beneficial, lightweight, application-specific static checking for Java.

## Acknowledgments

We would like to thank the anonymous reviewers for their helpful comments on an earlier version of this paper. This research was supported in part by NSF CCF-0430118.

## References

- [1] SourceForge. <http://www.sourceforge.net>.
- [2] SPEC JVM98 Benchmarks. <http://www.spec.org/jvm98/>.
- [3] Java Enterprise Edition HttpServletRequest API, 2006. <http://java.sun.com/javase/5/docs/api/javax/servlet/http/HttpServletRequest.html>.
- [4] A. Aiken, J. S. Foster, J. Kodumal, and T. Terauchi. Checking and Inferring Local Non-Aliasing. In *PLDI'03*, pages 129–140, June 2003.
- [5] C. Andreae, J. Noble, S. Markstrum, and T. Millstein. A framework for implementing pluggable type systems. In *OOPSLA'06*, pages 57–74, 2006.
- [6] K. Arnold, J. Gosling, and D. Holmes. *The Java Programming Language*. Addison-Wesley, 4th edition, 2006.
- [7] S. Artzi, M. D. Ernst, D. Glasse, and A. Kiezun. Combined static and dynamic mutability analysis. Technical Report MIT-CSAIL-TR-2006-065, MIT CSAIL, Sept. 2006.
- [8] A. Birka and M. D. Ernst. A practical type system and language for reference immutability. In *OOPSLA'04*, pages 35–49, Oct. 2004.
- [9] P. Broadwell, M. Harren, and N. Sastry. Scrash: A System for Generating Secure Crash Information. In *Usenix Security'03*, Aug. 2003.
- [10] B. Chin, S. Markstrum, and T. Millstein. Semantic type qualifiers. In *PLDI '05*, pages 85–95, 2005.
- [11] B. Chin, S. Markstrum, T. Millstein, and J. Palsberg. Inference of User-Defined Type Qualifiers and Qualifier Rules. In *ESOP'06*, pages 264–278, Mar. 2006.
- [12] J. Dean, D. Grove, and C. Chambers. Optimizatin of Object-Oriented Programs Using Static Class Hierarchy Analysis. In *ECCOP'95*, pages 77–101, Aug. 1995.
- [13] M. D. Ernst and D. Coward. JSR 308: Annotations on Java types. <http://jcp.org/en/jsr/detail?id=308>, July 2007.

- [14] J. S. Foster, M. Fähndrich, and A. Aiken. A Theory of Type Qualifiers. In *PLDI'99*, pages 192–203, May 1999.
- [15] J. S. Foster, R. Johnson, J. Kodumal, and A. Aiken. Flow-Insensitive Type Qualifiers. *ACM TOPLAS*, 28(6):1035–1087, Nov. 2006.
- [16] J. S. Foster, T. Terauchi, and A. Aiken. Flow-Sensitive Type Qualifiers. In *PLDI'02*, pages 1–12, June 2002.
- [17] T. E. Foundation. Eclipse Project. Web pages at <http://www.eclipse.org>.
- [18] T. Fraser, J. Nick L. Petroni, and W. A. Arbaugh. Applying flow-sensitive CQUAL to verify MINIX authorization check placement. In *PLAS'06*, 2006.
- [19] T. Freeman and F. Pfenning. Refinement types for ML. In *PLDI '91*, pages 268–277, 1991.
- [20] M. Furr and J. S. Foster. Polymorphic Type Inference for the JNI. In *ESOP'06*, pages 309–324, Mar. 2006.
- [21] V. Haldar, D. Chandra, and M. Franz. Dynamic Taint Propagation for Java. In *ACSAC'05*, pages 303–311, 2005.
- [22] A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM TOPLAS*, 23(3):396–450, 2001.
- [23] Java-Gnome Developers. Java bindings for the gnome and gtk libraries. <http://java-gnome.sourceforge.net>.
- [24] R. Johnson and D. Wagner. Finding User/Kernel Bugs With Type Inference. In *Usenix Security'04*, Aug. 2004.
- [25] M. S. Lam, J. Whaley, V. B. Livshits, M. C. Martin, D. Avots, M. Carbin, and C. Unkel. Context-sensitive program analysis as database queries. In *PODS'05*, pages 1–12, 2005.
- [26] O. Lhoták and L. Hendren. Scaling Java points-to analysis using Spark. In *CC'03*, pages 153–169, 2003.
- [27] O. Lhotak and L. Hendren. Jedd: a BDD-based Relational Extension of Java. In *PLDI'04*, pages 158–169, 2004.
- [28] S. Liang. *The Java Native Interface: Programmer's Guide and Specification*. Addison-Wesley, 1999.
- [29] Y. Liu and A. Milanova. Ownership and Immutability Inference for UML-based Object Access Control. In *ICSE'07*, pages 323–332, 2007.
- [30] B. Livshits, J. Whaley, and M. S. Lam. Reflection Analysis for Java. In *APLAS'05*, pages 139–160, 2005.
- [31] A. Milanova, A. Rountev, and B. G. Ryder. Parameterized Object Sensitivity for Points-to Analysis for Java. *ACM TOSEM*, 14(1):1–41, 2005.
- [32] R. Milner. A Theory of Type Polymorphism in Programming. *JCSS*, 17:348–375, 1978.
- [33] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. In *CC'02*, pages 213–228, Apr. 2002.
- [34] P. Pratikakis, J. Spacco, and M. Hicks. Transparent Proxies for Java Futures. In *OOPSLA'04*, pages 206–223, 2004.
- [35] W. Pugh. JSR 305: Annotations for Software Defect Detection, 2006. <http://jcp.org/en/jsr/detail?id=305>.
- [36] J. Rehof and M. Fähndrich. Type-Based Flow Analysis: From Polymorphic Subtyping to CFL-Reachability. In *POPL'01*, pages 54–66, Jan. 2001.
- [37] T. Reps. Undecidability of context-sensitive data-independence analysis. *ACM TOPLAS*, 22(1):162–186, 2000.
- [38] T. Reps, S. Horwitz, and M. Sagiv. Precise Interprocedural Dataflow Analysis via Graph Reachability. In *POPL'95*, pages 49–61, Jan. 1995.
- [39] A. Rountev. Precise Identification of Side-effect-free Methods in Java. In *ICSM'04*, pages 82–91, Sept. 2004.
- [40] B. G. Ryder. Dimensions of Precision in Reference Analysis of Object-oriented Programming Languages. In *CC'03*, pages 126–137, 2003.
- [41] A. Salcianu and M. Rinard. Purity and Side Effect Analysis for Java Programs. In *VMCAI'05*, Jan. 2005.
- [42] U. Shankar, K. Talwar, J. S. Foster, and D. Wagner. Detecting Format String Vulnerabilities with Type Qualifiers. In *Usenix Security'01*, Aug. 2001.
- [43] M. Sridharan and R. Bodik. Refinement-based Context-sensitive Points-to Analysis for Java. In *PLDI'06*, pages 387–400, 2006.
- [44] M. Sridharan, D. Gopan, L. Shan, and R. Bodik. Demand-driven Points-to Analysis for Java. In *OOPSLA'05*, pages 59–76, 2005.
- [45] R. E. Strom and S. Yemini. Tpestate: A programming language concept for enhancing software reliability. *IEEE Trans. Softw. Eng.*, 12(1):157–171, 1986.
- [46] M. S. Tschantz. Javari: Adding reference immutability to Java. Master's thesis, MIT Dept. of EECS, Aug. 2006. MIT-CSAIL-TR-2006-059.
- [47] M. S. Tschantz and M. D. Ernst. Javari: Adding reference immutability to Java. In *OOPSLA'05*, pages 211–230, Oct. 2005.
- [48] J. Whaley and M. S. Lam. Cloning-based Context-sensitive Pointer Alias Analysis using Binary Decision Diagrams. In *PLDI'04*, pages 131–144, 2004.
- [49] X. Zhang, A. Edwards, and T. Jaeger. Using CQUAL for Static Analysis of Authorization Hook Placement. In *Usenix Security'02*, Aug. 2002.