Practical pluggable types for Java

by

Matthew M. Papi

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2008

Author . . . .
Department of Electrical Engineering and Computer Science
May 23, 2008

Certified by . . . .
Michael D. Ernst
Associate Professor
Thesis Supervisor

Accepted by . . . .
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Practical pluggable types for Java

by

Matthew M. Papi

Submitted to the Department of Electrical Engineering and Computer Science
on May 23, 2008, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

This paper introduces the Checker Framework, which supports adding pluggable type systems to the Java language in a backward-compatible way. A type system designer defines type qualifiers and their semantics, and a compiler plug-in enforces the semantics. Programmers can write the type qualifiers in their programs and use the plug-in to detect or prevent errors. The Checker Framework is useful both to programmers who wish to write error-free code, and to type system designers who wish to evaluate and deploy their type systems.

The Checker Framework includes new Java syntax for expressing type qualifiers; declarative and procedural mechanisms for writing type-checking rules; and support for flow-sensitive local type qualifier inference and for polymorphism over types and qualifiers. The Checker Framework is well-integrated with the Java language and toolset.

We have evaluated the Checker Framework by writing five checkers and running them on over 600K lines of existing code. The checkers found real errors, then confirmed the absence of further errors in the fixed code. The case studies also shed light on the type systems themselves.

Thesis Supervisor: Michael D. Ernst
Title: Associate Professor

Acknowledgements

Michael Ernst provided invaluable guidance and discussion throughout the various stages of this research. Jeff Perkins used the Nullness checker, provided numerous bug reports, and suggested usability improvements. Mahmood Ali created the IGJ checker and contributed to the design and implementation of the Checker Framework. Telmo Correa created the Javari checker. Jaime Quinonez created the Annotation File Utilities. Thanks to my co-authors for their work on *Practical pluggable types for Java* [44], which has formed the basis of Chapters 4–6 of this thesis. Finally, thanks to my family and friends for their support and encouragement.

This work was supported in part by DARPA grant FA8750-06-2-0189.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

We present a practical three-part system for finding and preventing bugs in Java programs through user-defined extensions to Java's type system. Programmers can write type qualifiers, which provide extra information about the types of expressions, and can verify the correct use of these qualifiers by running a plug-in for a Java compiler. Programmers may design their own type qualifiers and create type checkers by extending our system, or they may use the type checkers for the qualifiers that we have defined.

The system has three parts:

1. A syntax for writing type qualifiers in Java programs, and a reference implementation of an extended Java compiler that accepts this syntax, exposes type qualifiers through the program's abstract syntax tree (AST), and persists type qualifiers by writing them to the class files that the compiler produces.

2. A flexible framework for writing type qualifier verifiers (also known as type checkers, or just checkers) that integrates tightly with the Java language toolchain. Its features include a means for both declarative and procedural specification of type rules and type qualifier introduction, polymorphism for types and qualifiers, flow-sensitive intraprocedural type qualifier inference, and a representation of annotated types.

3. Five checkers: four for specific type systems and one for any type system that has

no semantics beyond standard Java subtyping rules. These particular type qualifier systems are useful to programmers; they are non-trivial to implement, serving as useful evaluation of the framework; and in some cases, they are the only existing verifier implementation and serve to evaluate the type qualifier system itself.

1.1 Terminology

This section introduces the terms used throughout this thesis.

A *type* determines the set of values that a variable may contain and the operations that may be performed on that variable. Examples of types include the composite type `File`, which is used for variables that represent files in a filesystem, or the primitive type `boolean`, which is used for variables that have either the value "true" or the value "false".

A *type qualifier* extends a type with a particular optional attribute. An example of a type qualifier is `encrypted`, which denotes that a value is securely encrypted. When qualifying the type `File`, the result is a new type, `encrypted File`; the relationship between `encrypted File` and `File` is part of the semantics of the `encrypted` type qualifier. We write type qualifiers in two ways. When discussing type qualifiers in general, we use a form similar to the keywords of many programming languages: `encrypted`. When discussing specific type qualifiers in the context of our system, write them according to our proposed Java syntax: `@Encrypted`.

A *type system* defines how a programming language classifies the types of data in a program and describes how these types interact. The Java type system includes all of the composite, primitive, and user-defined types in the Java language and specifies the rules of subtyping, assignment, etc.

A *type qualifier system* defines a set of type qualifiers and the interactions between them. The Nullness type qualifier system of Section 5.3.1 defines the `nonnull` and `nullable` qualifiers, specifies that for any type `T`, `nonnull T` is a subtype of `nullable T`, and requires that no reference with type `nullable T` may be dereferenced.

A *type checker* is an algorithm or a tool that verifies that a program does not violate the rules of a type system. A *type qualifier verifier*, which we refer to as a *type qualifier* *checker* or more commonly as a *type checker* in the context of qualifiers, is a type checker that verifies that a program does not violate the rules of a type qualifier system.

A type qualifier system is *pluggable* if it is optional, after Bracha [9]. We often describe a type checker as pluggable to mean that it can be invoked as a "plug-in" by a compiler during compilation (i.e., that it is not built in to the compiler, but is dynamically loaded as a separate module); note that this is a separate sense of the word "pluggable".

1.2 Thesis outline

The remainder of this thesis is organized as follows.

Chapter 2 discusses the importance of type systems and the benefits of using type qualifiers, and provides an example scenario for detecting security-related bugs in a program.

Chapter 3 describes the syntactic means we have created for writing type qualifiers in Java programs. The syntax follows the JSR 308 specification for annotations on types. The design of the JSR 308 specification was motivated in part by this work and our desire to have most of the benefits and few of the drawbacks of previous approaches. We have created the reference implementation for the JSR 308 specification, described in Section 3.4, and use it as the foundation for our system.

Chapter 4 describes a framework for creating type checkers and how it may be used to implement the semantics of a type qualifier system. The framework, known as the Checker Framework, sits atop the JSR 308 Java compiler: type checkers integrate closely with the Java compiler, and programmers write type qualifiers as Java annotations using the JSR 308 syntax. The Checker Framework provides the essential functionality for writing a pluggable type checker — an interface to the Java compiler and a facility for determining the qualified type of a program element or expression — as well as a number of powerful optional features that can make type qualifier systems easier to use and more expressive, like type qualifier polymorphism, default annotations, and flow-sensitive intraprocedural type qualifier inference.

Chapter 5 describes the checkers we have built using the framework. There are type checkers for four distinct type systems: the Nullness type system for finding and preventing null pointer errors, the Interning type system for finding and preventing interning and equality-testing errors, and the IGJ and Javari type systems for finding and preventing mutability errors (based on the IGJ [57] and Javari [51] languages). We have also created a type checker that can be applied to any type system that does not have special semantics beyond standard Java subtyping rules. We have evaluated the checkers by using them to find bugs in 13 programs of up to 224 KLOC, including the checkers themselves.

Chapter 6 discusses related work.

Chapter 7 describes possibilities for future work, and summarizes the contributions of this research and lessons we have learned while conducting it.

Finally, Appendix A provides a source code listing for the implementation of the Interning checker described in Section 5.4.

Chapter 2

Motivation

Type systems — a means for specifying the classification of and interactions between data in a program — are an important part of many modern programming languages. Computer hardware stores and manipulates program data as groups of bits, regardless of whether these bits represent integers, character strings, or other data structures. From the point of view of a programmer or a programming language, these data are distinctly different. Moreover, they *must* be treated as such — it does not make sense to multiply a character string and a list of floating-point values.

Type systems allow programmers to organize and document data according to their types. This both improves a programmer's understanding of his own program and helps communicate his intent to other programmers and program maintainers.

A static type system requires that variables have a type that does not change throughout a program. Compilers typically use type information to check a program for type errors, in which data is used nonsensically (for example, attempting to multiply a string and a list of floating-point values). These errors can be detected and fixed before running the program. Compilers for statically-typed languages *prevent* type errors from occurring in programs, since they reject any program that contains type errors.

Type information can be used in other ways as well: an optimizing compiler could replace calls to virtual functions with direct function calls, or produce machine code that utilizes specialized instructions such as those for string-processing or floating-point math. Type information also facilitates many of the conveniences provided by IDEs, including refactorings and automatic code completion.

Despite the strengths of modern type systems, there is often a great deal of information about data that a programmer cannot specify. Java has a data type for specifying that a value represents a string of characters (via the type `String`), but it has no provision for specifying other properties of the type of that value — for instance, whether the `String` is encrypted, or represents XML or a SQL query. In the same way that only `String`s can be written to a file, only encrypted `String`s should be used to transmit secret data over a network, and only `String`s that represent XML should be given to an XML parser.

This extra information is typically known to the programmer ahead of time, and it is usually conveyed through comments or other forms of documentation. Comments can be logically inconsistent (e.g., "this integer is nonnegative" versus "this integer is never zero or less"), and programmers often forget to update them when the specification changes (or they forget to write them in the first place). Worse, comments convey nothing to tools like compilers that could use this extra information to enhance error-checking and optimization.

When constraints are specified via type refinements, they can be statically verified. The programmer learns of errors instantly (literally, if he is using a development environment that supports this!) and can fix them before running the program. More importantly, if the compiler emits no errors, the programmer receives a guarantee that the program will not contain errors related to these constraints.

2.1 Example: preventing security-related bugs

To demonstrate the value of using type qualifiers, consider the hypothetical `encrypted` qualifier. `encrypted` extends the typical notion of Java types to denote a special version of a type for values with an encrypted representation. For instance, the type `String` is used to describe a sequence of characters that may or may not be encrypted; the addition of the `encrypted` qualifier results in a new type, `encrypted String`, that is used to describe an encrypted sequence of characters. The Java type `Object` is a subtype of `String`: every `String` is an `Object`, but not every `Object` is a `String`. Similarly, the qualified type `encrypted String` is subtype of `String`: every `encrypted String` is a `String` but not every `String` is an `encrypted String`.

We can use the `encrypted` qualifier to enhance the type of data that we know to be encrypted. For programs for which security is a concern, this has several advantages. First, we document in a clear and concise way which methods (for instance) require encrypted inputs, or which lists contain only encrypted data. Developers and program maintainers need only to read the method's signature (its name plus the types of its formal parameters and return value) to learn how that method is used in the context of security via encryption.

Second, and more importantly, a tool can verify that only data that is guaranteed to be encrypted is passed to methods that require an encrypted input or added to lists that only allow encrypted data. This means that some security bugs (like those caused by using unencrypted data where encrypted data is required) can be detected program compilation and immediately reported to the programmer. As a corollary, the programmer receives a guarantee that any program that type-checks in the hybrid Java-plus-`encrypted` type system does not contain any such security bugs.

Finally, as a minor point, the use of the `encrypted` qualifier can improve program simplicity and performance: programs that use `encrypted` can eliminate runtime security checks, such as calls to a verification method that ensures that data at a particular program point is encrypted; type-checking of `encrypted` makes these runtime checks completely redundant.

To illustrate the use of an `encrypted` qualifier, we examine its use as part of a hypothetical program for communicating securely via a network.

Figure 2-1 shows a fragment of a Java program for sending messages over a network as part of a chat program. The program has a `sendOverNetwork` method that is used to send strings of text to other machines on a network; a comment on that method explains that its callers should first encrypt messages before sending them. Note that, other than by manual inspection, this recommendation is not enforced. As a result, a programmer introduces a bug in which a password is given to `sendOverNetwork` without any encryption.

Figure 2-2 demonstrates the use of an `encrypted` qualifier — written in the program as a Java annotation, `@Encrypted` — which permits a type checker to reveal the bug in the program. `@Encrypted` is added to the `message` parameter of the `sendOverNetwork` method. This qualifier effectively restricts the input of the method in the same way that the comment suggests; in this case, however, the restriction is statically checkable. A type checker for `@Encrypted` would notice that `password`, as passed to the method invocation on line 17, is not encrypted (since it was not declared as such on line 16), and the checker would emit an error accordingly.

Note that simply adding an `@Encrypted` annotation to the declaration on line 16 is not sufficient for fixing the problem unless the `getUserPassword` method returns an encrypted password (and its return type has been accordingly annotated as `@Encrypted`). A proper fix would involve calling the `encrypt` method on the password and passing the result to `sendOverNetwork`; as is evident from its signature, the `encrypt` method takes a message and returns it in encrypted form.

The addition of `@Encrypted` annotations made it possible to detect security-related bugs. It also serves as better (and much more concise) documentation than the code comment in lines 3–4 of Figure 2-1.

```
1   class ChatProgram {
2
3     // This method performs no encryption! Callers must encrypt
4     // messages before calling this method.
5     void sendOverNetwork(URI destination, String message) {
6       ...
7     }
8
9     // Returns the encryption of ``message''.
10    String encrypt(String message) {
11      ...
12    }
13
14    void logIn() {
15      ...
16      String password = getUserPassword();
17      sendOverNetwork(server, password);
18    }
19  }
```

Figure 2-1: A fragment of a network chat program with a security bug.

```
1   class ChatProgram {
2
3
4
5     void sendOverNetwork(URI destination, @Encrypted String message) {
6       ...
7     }
8
9     // Returns the encryption of ``message''.
10    @Encrypted String encrypt(String message) {
11      ...
12    }
13
14    void logIn() {
15      ...
16      String password = getUserPassword();
17      sendOverNetwork(server, password);
18    }
19  }
```

Figure 2-2: The fragment of a network chat program in Figure 2-1 with `@Encrypted` annotations added. The annotations enable a type checker for `@Encrypted` to find the security bug: on line 17 a value of type `String` is passed to a method that expects a parameter of type `@Encrypted String`.

Chapter 3

Syntax: Annotations on types

The Checker Framework's syntax for type qualifiers uses the proposed JSR 308 specification for writing Java annotations anywhere that types are used; the system itself employs a modified Java compiler that we have created to support the JSR 308 syntax.

This chapter discusses our syntax design choice (Section 3.1), presents the syntax for annotations on types (Section 3.2), describes the use of the JSR 308 Java compiler (Section 3.3), and describes our system's implementation of the JSR 308 specification (Section 3.4).

3.1 Design Rationale

The Checker Framework uses Java annotations to express type qualifiers. This section presents the rationale behind this design choice and compares it with those of related tools. We briefly discuss three approaches to type qualifier syntax: language keywords, stylized code comments, and annotations.

Language keywords Some previous implementations for specific, non-pluggable type systems have modified the language compiler to add new keywords (syntactically similar to Java keywords like `transient` or `final`) [5]. The advantage to this approach is that the type system is fully integrated with the language, making it easy for programmers and analysis tools to use. The disadvantage is that it is difficult to implement and non-portable — other conforming Java compilers and tools such as IDEs must be reimplemented to support the changes.

Stylized code comments Other systems use code comments with a special format [29]. The approach requires no changes to the language toolchain, and therefore is completely portable. However, since the compiler ignores code comments, this approach is not the most robust; in contrast, compilers and IDEs can emit an error for unrecognized or misplaced language keywords and annotations.

Annotations Annotations were introduced in Java 5 [7] as a way of expressing program metadata. For instance, a method may have a `@Deprecated` annotation to denote that the method should not be used, or a class might have an `@Author` annotation to specify the creator of the class. Annotations are part of Java's syntax but are user-definable; they combine the power of language keywords with the flexibility of code comments.

However, standard Java annotations are not powerful enough for use as type qualifiers; in Java 5 and 6, annotations are permitted *only* on the declarations of classes, fields, methods, method parameters, and local variables. There are many places where types are used but annotations are disallowed: type casts, generic type arguments, type variable bounds, and array creation expressions name a few. We have defined an extension that permits annotations to appear on nearly any use of a type [23]. Examples of the new syntax are:

```
List<@NonNull String> strings;
myGraph = (@Immutable Graph) tmpGraph;
class UnmodifiableList<T>
  implements @ReadOnly List<@ReadOnly T> { ... }
```

Some systems use special annotations to resolve some of these issues, but to date no other annotation-based proposal exists for writing type qualifiers *everywhere* that a type can be used.

Our syntax proposal has been assigned the Sun codename "JSR 308" [23] and is planned for inclusion in the Java 7 language. These simple changes to Java enable the construction of a type-checking framework, described in Chapter 4, that requires no compiler changes beyond those planned for inclusion in Java 7.

Changing the Java language is extraordinarily difficult for technical reasons largely revolving around backward compatibility, but is worth the effort if practical impact is the goal. Workarounds are clumsy and inexpressive. For example, stylized comments are not recognized by tools such as IDEs and refactoring engines; by contrast, our implementation works with the NetBeans IDE and Jackpot transformation engine. A separate tool is rarely as robust as the language compiler, but directly modifying a compiler results in an incompatible system that is slow to incorporate vendor updates. Programmers are unlikely to embrace these approaches.

3.2 Writing annotations on types

JSR 308 [23] proposes a syntax that permits annotations to appear on nearly any use of a type. JSR 308 uses a simple prefix syntax for type annotations, with three exceptions necessitated by non-orthogonalities in the Java grammar. The changes to the Java language grammar are:

1. A type annotation may be written before any type, as in `@NonNull String`.
2. A method receiver (`this`) type is annotated just after the parameter list.
3. An array type is annotated on the brackets `[]` that indicate the array, separately from an annotation on the element type.
4. The component type of a variable-argument (varargs) method parameter is annotated immediately before the ellipsis (...).

Figure 3-1 shows the changes to the Java grammar in detail; these changes correspond to the four rules above.

The following examples show the use of annotations on a variety of types, none of which are valid locations for Java 5 and 6 annotations:

Type:
  [Annotations] *Identifier [TypeArguments]* { . *Identifier [TypeArguments]*} {`[` *[Annotations]* `]`}
  [Annotations] *BasicType*

VoidMethodDeclaratorRest:
  FormalParameters *[Annotations]* [`throws` *QualifiedIdentifierList*] (*MethodBody* | `;`)

FormalParameterDeclsRest:
  VariableDeclaratorId [, *FormalParameterDecls*]
  [Annotations] ... *VariableDeclaratorId*

Figure 3-1: A summary of the changes to the Java grammar for JSR 308. Additions are underlined.

- generic type arguments to parameterized classes:

```
Map<@NonNull String, @NonEmpty List<@ReadOnly Document>> files;
```

- generic type arguments in a generic method or constructor invocation:

```
o.<@NonNull String>m("...");
```

- type parameter bounds and wildcards:

```
class Folder<F extends @Existing File> { ... }
Collection<? super @Existing File>
```

- class inheritance:

```
class UnmodifiableList<T>
 implements @ReadOnly List<@ReadOnly T> { ... }
```

- method `throws` clauses:

```
void monitorTemperature()
  throws @Critical TemperatureException { ... }
```

- typecasts:

```
myString = (@NonNull String) myObject;
```

- type tests:

```
boolean isNonNull = myString instanceof @NonNull String;
```

- object creation:

```
new @NonEmpty @ReadOnly List<String>(myNonEmptyStringSet)
```

- method receivers:

```
public String toString() @ReadOnly { ... }
public void write() @Writable throws IOException { ... }
```

- class literals:

```
Class<@NonNull String> c = @NonNull String.class;
```

- static member access:

```
@NonNull Type.field
```

- arrays:

```
Document[@ReadOnly][] docs1 = new Document[@ReadOnly 2][12];
Document[][@ReadOnly] docs2 = new Document[2][@ReadOnly 12];
```

These additional locations allow the Checker Framework to retain tight integration with the Java language without sacrificing expressiveness.

JSR 308 also specifies a means for persisting type annotations by writing them to a class file. Section 3.4 discusses this in greater detail.

3.3 Using the JSR 308 compiler

3.3.1 Invoking the Java compiler

The JSR 308 Java compiler can be used in exactly the same way as a standard Java compiler. The modifications to the OpenJDK compiler concern only the JSR 308 specification and fixes to bugs in the unmodified compiler to support JSR 308.

The OpenJDK compiler provides the `-processor` command-line switch, which is used to invoke annotation processors during program compilation. The JSR 308 compiler adds a `-typeprocessor` switch that is featurewise identical to `-processor` but reserves the possibility for additional functionality in the JSR 308 compiler to support type checking of qualifiers.

3.3.2 Backward-compatibility

The JSR 308 Java compiler permits the use of annotations on types while retaining backward compatibility with previous versions of the Java toolchain that do not support JSR 308. Annotations on types can be written using a special syntax which the JSR 308 Java compiler parses as an annotation but non-JSR 308 Java compilers parse as comments. For example, a non-JSR 308 compiler will reject the following program fragment:

```
void deleteFiles(List<@NonNull File> files) {
  // ...
}
```

Since the `@NonNull` annotation is not written in a location permitted by standard Java annotations, a standard Java compiler will emit a parse error. However, a non-JSR 308 compiler will accept the following code fragment:

```
void deleteFiles(List</*@NonNull*/ File> files) {
  // ...
}
```

A JSR 308 compiler, on the other hand, constructs an abstract syntax tree that is identical for the two program fragments above and includes the `@NonNull` annotation.

3.3.3 Examining class files with JSR 308 annotations

We have also augmented the `javap` tool, a disassembler for displaying the contents of Java class files in a human-amenable form.

The standard OpenJDK `javap` tool displays the structure of a class file, including classes, fields, and methods, and attributes of those program elements. It does not display detailed information about either standard[1] or JSR 308 annotations, instead giving a hexadecimal representation of their containing attributes.

Enhancements to the `javap` tool for JSR 308 permit the tool to display detailed information about the types, arguments, and location of both standard and JSR 308 annotations.

3.4 JSR 308 reference implementation

The JSR 308 `javac` compiler is implemented as a modification of Sun's OpenJDK `javac` compiler[2]. By relying on the existing `javac` support for standard Java annotations, our modifications are generally straightforward and robust.

The OpenJDK Java compiler, which is written in Java, is used to transform Java source files into class files for execution in the Java virtual machine. The OpenJDK compiler is an open-source version of Sun's proprietary Java compiler. Our modifications are publicly redistributed[3] under an open-source license.

Our extensions to the compiler must support the four primary features that the Java compiler provides for standard Java annotations:

- *parsing* annotations and adding them to the abstract syntax tree (AST) of an input program;
- *resolving* (also referred to as *entering*) annotations present in the AST, which includes locating the annotation's definitions, checking uses of the annotations for errors, and reducing them to a simpler representation for later compilation;
- *writing* the annotations into the class file, so that they can be read back during compile time or runtime; and
- *reading* the annotations from a class file, so that they can be used when compiling against programs for which the source code is not available.

[1](at the time of this writing)
[2]`http://openjdk.java.net`
[3]`http://pag.csail.mit.edu/jsr308`

The following sections describe the implementation of these four features for JSR 308 annotations.

3.4.1 Parsing JSR 308 annotations

Since standard Java annotations are only permitted in the same places as modifier keywords (e.g., `public`, `static`, `final`), the standard Java compiler represents annotations (as well as modifier keywords) in an AST node called a `ModifiersTree`. Reuse of a `ModifiersTree` for extended annotations is possible but is not good semantics, since only extended annotations (and not modifier keywords) should be permitted on types.

The extended compiler introduces a new AST node, the `AnnotatedTypeTree`, which combines of a list of annotation nodes and a node that represents a type. (Since the standard compiler does not have a special designation for trees that represent types, the `AnnotatedTypeTree` has two members, one of type `List<AnnotationTree>` for storing annotations, and another of type `Tree` to point to the underlying type of the annotated type.)

The parser then creates an `AnnotatedTypeTree` for all locations where annotations may be written on a type. For instance, when parsing the expression `List<String>`, the standard Java parser would create a `ParameterizedTypeTree` to represent `List`, and add as a child a `IdentifierTree` to represent `String`. Since extended annotations may be written on `List` (e.g., `@NonNull List<String>`), the parser first creates an `AnnotatedTypeTree`, then parses annotations before the type (adding them as the annotations of the `AnnotatedTypeTree`), and finally parses the type itself (adding it as the underlying type of the `AnnotatedTypeTree`).

Extended annotations can be written on type arguments (e.g., `List<@NonNull String>`), so the parser also creates an `AnnotatedTypeTree` for each type argument. That is, it creates an `AnnotatedTypeTree`, then parses any annotations before `String` and adds them to the `AnnotatedTypeTree`, then parses the type and adds it as the underlying type. Finally, it adds this `AnnotatedTypeTree` as the child of the `ParameterizedTypeTree` for `List`. Figure 3-2 shows the AST that the compiler creates when parsing the expression `@NonNull List<@NonNull String>>`.

The modified parser abstracts away the process of parsing annotations on a type via the

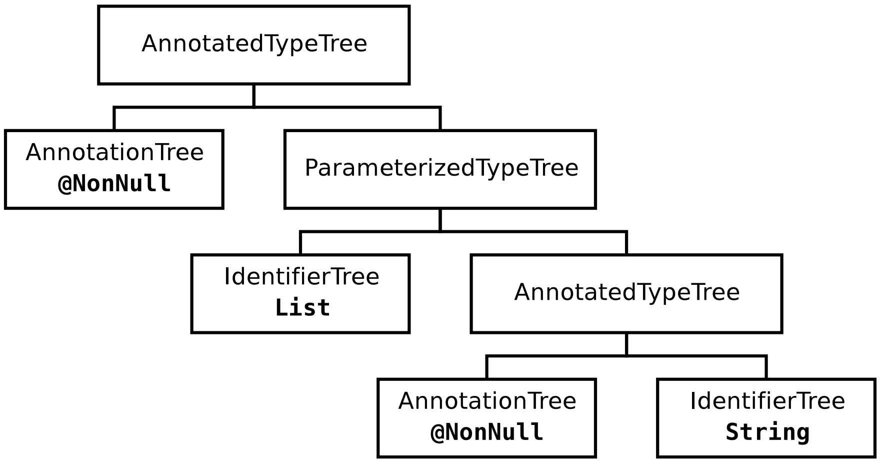

Figure 3-2: The AST for the expression `@NonNull List<@NonNull String>`.

`typeAnnotationsOpt` method (from "optional type annotations"), which parses annotations on a type and returns a possibly empty list of `AnnotationTree`s. The modifications to the parser consist mostly of calls to `typeAnnotationsOpt` and construction of `AnnotatedTypeTree`s from the resulting annotations and the underlying type; exceptions are noted below.

Difficulties

There are two areas that introduce complexity in the modifications to the parser.

AST non-orthogonalities Parse trees for array expressions, which are represented via `ArrayTypeTree`, are hierarchical. For instance, when constructing a parse tree for the type `File[][]` (an array of arrays of `File`s), the compiler creates an `ArrayTypeTree` which has a child node that is an `ArrayTypeTree`, which itself has a child node that is an `IdentifierTree` for the identifier `File`. The parse tree that is constructed is structurally similar to the type itself: an array type that has another array type as a component.

Parse trees for array creation expressions, however, are not always hierarchical. For instance, consider the following expression:

```
new File[][] { ... }
```

This expression creates a new array and initializes it to the contents of the braces ("{" and "}"). Its parse tree consists of a `NewArrayTree`, which has a child node that is an `ArrayTypeTree`, which has a child node that is an `IdentifierTree` for `File`. Here, the `NewArrayTree` implicitly represents one level of the array, replacing the outermost `ArrayTypeTree` in case of the parse tree for `File[][]`.

Furthermore, in the case of an array creation with explicit dimensions (e.g., `new File[5][10]`), the child of the `NewArrayTree` is simply an `IdentifierTree` and expresses no hierarchy. Instead, the hierarchy can be determined via a list of dimensions.

Due to the unique way in which `NewArrayTree`s are constructed, the JSR 308 implementation enhances the `NewArrayTree` by storing dimension annotations in a list alongside the dimensions themselves.

Array conventions Second, different conventions for annotating array types complicate parsing independently from the non-orthogonalities described above.

The AST has a single semantic meaning, but annotated array expressions in the program do not. As an example, consider that for an array of `Document`s (written in Java as `Document[]`), there are two types that a programmer may wish to annotate:

- the array type as a whole (of type `Document[]`), e.g. to specify a `@NonNull` array of `Document`s
- the type of elements in the array (of type `Document`), e.g. to specify an array of `@NonNull Document`s

There are two different conventions for annotating an array type. In one, the *arrays* convention, annotations within brackets refer to the array corresponding to those brackets; in the other, the *elements* convention, the annotations within brackets refer to the elements of the array corresponding to those brackets. Under these conventions, we write "array of `@NonNull Document`s" as follows:

- *arrays*: `@NonNull Document[]`

- *elements*: `Document[@NonNull]`

It is important to note that, regardless of convention, there is *exactly one* parse tree that corresponds to "array of `@NonNull Documents`": namely, an `ArrayTypeTree`, which has as a child an `AnnotatedTypeTree` with a child for `@NonNull` and a child for `Document`. As a consequence, the array convention is determined by the parser in the JSR 308 compiler.

The *elements* convention is simpler to implement: parsing occurs from left to right, and annotations written under this convention have spacial locality with the types they annotate. For instance, the leftmost annotation belongs on the type of the entire array, while the subsequent annotations are added to the intermediate array levels in the parse tree.

For the *arrays* convention, however, the leftmost annotation belongs in the deepest level of the parse tree for the array type, while the rightmost annotation belongs on the outermost level of the parse tree for the array type. This makes parsing array types under the *arrays* convention require additional tree manipulation and traversal that is not needed for the *elements* convention.

3.4.2 Resolving annotation locations

Resolution of JSR 308 annotations is identical to that of resolution of standard annotations, except that JSR 308 annotations require an extra step to determine the precise location of the annotation in the AST.

Standard annotations appear in *attributes* in Java class files; attributes are "attached" to structural elements in the class file: classes, methods, method parameters, and fields. The possible locations of attributes correspond exactly to the possible locations of standard annotations.

Since types do not have a direct representation in a class file, JSR 308 requires that the annotations on a type appear in an attribute that is attached to the nearest enclosing program element for the type. For instance, if the annotation appears on the type in a typecast in the body of a method, the annotation must be stored in the class file in an attribute of that method. Likewise, if the annotation appears on a type argument in the declaration of a field with a generic type, the annotation must be stored in an attribute of that field.

Since these annotations are not written directly on (for instance) a method or field, extra information must be stored alongside the annotation so that its exact original location — the type that it was written on — can be determined when reading back the class file. Therefore, the JSR 308 compiler must determine the annotation's exact location. It does this in two rounds: first, when annotations are entered, and second, during the compiler's code generation phase. This second round is required only because JSR 308 requires bytecode offsets as part of the extra information for some annotations, and this information is unavailable until code generation.

As an example, an annotation on a typecast can be resolved as follows:

- In the first round, the compiler traverses the AST from the root searching for extended annotations. As it traverses the tree, it pushes each node that it visits onto a stack; it then descends into that node's children and pops the node from the stack when all its children have been visited. The stack forms a "path" of all the parent nodes of a tree up to the root. When it encounters an extended annotation, it examines the path to determine the kind of the annotation. In the case of a typecast annotation, the compiler notices that the annotation is immediately enclosed by a typecast node. The compiler then assigns the annotation a "target type" — in this case "typecast annotation" — and associates with it a pointer to the relevant enclosing tree (the immediately enclosing typecast node).

- In the second round, which occurs simultaneously with code generation, the compiler again traverses the AST from the top down, emitting bytecodes corresponding to each node that it visits. When the compiler reaches a typecast, it performs code generation as usual (in this case, emitting a `checkcast` bytecode), and it then looks for annotations with a context node equivalent to the node it is currently visiting. If one is found, the compiler can then assign the annotation extra location information — in this case, the offset of the `checkcast` bytecode emitted for the typecast — as required by the JSR 308 specification.

3.4.3 Writing annotations to the class file

JSR 308 annotations are written to the class file in a similar way to standard Java annotations; there are two major differences.

First, standard annotations are written to the `RuntimeVisibleAnnotations` and `RuntimeInvisibleAnnotations` attributes (and also the `RuntimeVisibleParameterAnnotations` and `RuntimeInvisibleParameterAnnotations` attributes for method parameters), but JSR 308 requires annotations to be written to the `RuntimeVisibleExtendedAnnotations` and `RuntimeInvisibleExtendedAnnotations` attributes. (We note that the Java virtual machine ignores unrecognized attributes in class files, so no modifications are needed to the JVM to support this.)

Second, extended annotations are written with location information, so that their precise location on a type within a class, method, method parameter, or field declaration can be recovered by an analysis tool that operates on bytecode instead of source code (e.g., a bytecode verifier).

3.4.4 Reading annotations from the class file

The compiler must only read back JSR 308 annotations that are public-facing. For instance, when a program refers to a method defined in class file (for which the source code is not available), the program examine anything in the body of the method, including annotations written on the types therein. The program only requires the method's signature to compile successfully, and therefore tools that utilize JSR 308 annotations only need the annotations on types in the method's signature. As a result, the compiler's class reader ignores annotations that are not on classes, fields, or method signatures. When reading the annotations, the compiler leaves their location information intact. The compiler does not provide a representation for annotated types; tools that require the use of these annotations can use the

Checker Framework described in Chapter 4.

Chapter 4

Semantics: The Checker Framework

The Checker Framework enables a type system designer to define the rules for a type qualifier system. The Checker Framework then creates a type-checking compiler plug-in (also known as a checker) that applies these rules to an input program during compilation. This chapter describes the aspects of the Checker Framework that support this process: how a programmer uses a checker to find errors in a program (Section 4.1), the architecture of a type system (Section 4.2) and the corresponding implementation of a checker (Sections 4.3–4.5), and advanced functionality that the Checker Framework provides to type system designers (Sections 4.7–4.8).

4.1 The programmer's view of a checker

This section describes the Checker Framework from the point of view of a programmer that wishes to find errors (or verify their absence) in a program. Section 4.2 describes it from a type system designer's point of view.

The Checker Framework seamlessly integrates type-checking with the compilation cycle. Programmers add qualifiers to types in their programs using the backwards-compatible extension to Java's annotation syntax described in Chapter 3. A checker runs as a plug-in to the `javac` Java compiler.

4.1.1 Using a checker to detect software errors

Type checkers built atop the Checker Framework are a special case of annotation processors [14]. The Checker Framework uses Java's standard compiler flag, `-processor`, for invoking an annotation processor:

```
javac -processor NullnessChecker MyFile.java
```

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.

The checker reports warnings and errors through the same standard reporting mechanism that the compiler itself uses (the public Compiler API, also known as JSR 199[55, 14]). As a result, checker errors/warnings are formatted like compiler errors/warnings, and the compiler is aware of checker errors/warnings when determining whether to continue compilation (e.g., perform code generation).

Use of `@SuppressWarnings` annotations and command-line arguments permits suppressing warnings by statement, method, class, or package. Naturally, the checker's guarantees that code is error-free apply only to analyzed code. Additionally, the framework does not reason about the target of reflective calls.

4.2 Architecture of a type system

The implementation of a type system contains four components:

1. **Type qualifiers and hierarchy.** Each qualifier restricts the values that a type can represent. The hierarchy indicates subtyping relationships among qualified types (for instance, that `@NonNull Object` is a subtype of `@Nullable Object`.)

2. **Type introduction rules.** For some types and expressions, a qualifier should be treated as present even if a programmer did not explicitly write it. For example, every literal (other than `null`) has a `@NonNull` type.

3. **Type rules.** Violation of the type system's rules yields a type error. For example, every assignment and pseudo-assignment must satisfy a subtyping rule. As another example, in the Nullness type system, only references with a `@NonNull` type may be dereferenced.

4. **Interface to the compiler.** The compiler interface indicates which annotations are part of the type system, the checker-specific compiler command-line options, which `@SuppressWarnings` annotations the checker recognizes, etc.

Sections 4.3–4.6 describe how the Checker Framework supports defining these four components of a type system. The Checker Framework also supports parametric polymorphism over both (qualified) types and type qualifiers (Section 4.7) and flow-sensitive inference of type qualifiers (Section 4.8). Source code listings from the implementation of the Interning checker (Section 5.4) are provided in Appendix A as a detailed example of the use of the Checker Framework.

The Checker Framework offers both declarative and procedural mechanisms for implementing a type system. The declarative mechanisms are Java annotations that are written primarily on type qualifier definitions; these extend the default functionality in the most common ways that we have encountered. The procedural mechanisms are a set of Java APIs that implement the default functionality; in most cases, a type system designer only needs to override a few methods. Because both mechanisms are Java, they are familiar to users and are fully supported by programming tools such as IDEs; a type system designer need not learn a new language and toolset. Users found the checker implementations clear to read and write.

Our experience and that of others [3] suggests that procedural code is essential when defining a realistic type checker, at least in the current state of the art. Our design also permits a checker to use specialized types and rules, even ones that are not expressible in the source code for reasons of simplicity and usability. Examples include dependent types, linear types, and reflective types.

A more important benefit is expressiveness: while simple type systems are concise, complex ones are possible. Parts of the implementations of the checkers described in Chapter 5 required sophisticated processing that no framework known to us directly supports. One example that requires sophisticated processing is that of the `Collection.toArray` method, which has a reflective, generic type.

The Checker Framework also provides a representation of annotated types, `AnnotatedTypeMirror`, that extends the standard `TypeMirror` interface of the Annotation Processing API [14] (JSR 269) with a representation of the annotations. As code uses all or part of a compound type, at every step the relevant annotations are convenient to access. This is particularly important for generic and array types: generic type arguments, array component types, and the bounds of wildcards and type parameters are themselves `AnnotatedTypeMirrors`. As a result, type system designers can express operations over annotated types concisely using the visitor design pattern or recursive procedures.

4.3 Type qualifiers and hierarchy

Type qualifiers are defined as Java annotations [14], extended as described in Chapter 3. A type system designer uses the `@TypeQualifier` meta-annotation to distinguish an annotation that represents a qualifier (e.g., `@NonNull` or `@Interned`) from an ordinary annotation (e.g. `@Deprecated` or `@Override`).

The type hierarchy induced by the qualifiers can be defined either declaratively (via meta-annotations) or procedurally. Declaratively, the type system designer writes a `@SubtypeOf` meta-annotation on the declaration of type qualifier annotations. (Java forbids annotations from subtyping one another.) `@SubtypeOf` accepts multiple annotation classes as an argument, permitting the type hierarchy to be an arbitrary DAG. For example, Figure 5-4 shows that in the IGJ type system (Section 5.6.1), `@Mutable` and `@Immutable` induce two mutually exclusive subtypes of the `@ReadOnly` qualifier.

While the declarative syntax suffices for many cases, more complex type hierarchies can be expressed by overriding the framework's `isSubtype` method, which is used to determine whether one qualified type is the subtype of another. The IGJ and Javari checkers specify the qualifier hierarchy declaratively, but the type hierarchy procedurally. In both type systems, some type parameters are covariant (with respect to qualifiers) rather than invariant as in Java. For example, in IGJ a `@ReadOnly List` of `@Mutable Date`s is a subtype of a `@ReadOnly List` of `@ReadOnly Date`s.

The `@DefaultQualifier` meta-annotation indicates which qualifier implicitly appears on unannotated types. This may ease the annotation burden (by reducing the number of annotations that must be written) or provide backward compatibility with unannotated programs.

A type system whose default is not the root of the qualifier hierarchy (such as `@ReadOnly` in Javari and IGJ) requires special treatment of `extends` clauses. The framework treats the declaration `class C<T extends Super>` as `class C<T extends` *`RootQual`* `Super>` if the class has no methods with a receiver bearing a subtype qualifier, and as `class C<T extends` *`DefaultQual`* `Super>` otherwise. This rule generalizes to hierarchies more complex than 2 qualifiers, and ensures backward compatibility while maximizing the number of possible type parameters that a client may use.

4.4 Implicit annotations: qualifier introduction

Certain constructs should be treated as having a type qualifier even when the programmer has not written one. For example, string literals are non-null and the JVM automatically interns them. Therefore, in the Nullness type system (Section 5.3.1), string literals implicitly have the type `@NonNull String`. In the Interning type system (Section 5.4), they implicitly have the type `@Interned String`.

Type system designers can specify this declaratively using the `@ImplicitFor` meta-annotation. It accepts as arguments up to three lists, of types (such as primitives or array types), symbols (such as exception parameters), and/or expressions (such as string literals) that should be annotated.

Type system designers can augment the declarative syntax by additionally overriding the Checker Framework's `annotateImplicit` method to apply implicit annotations to a type in a more flexible way. For instance, the Interning checker overrides `annotateImplicit` to apply `@Interned` to the return type of `String.intern`[1].

Implicit annotations are distinct from, and take precedence over, the default annotations of Section 4.3.

Implicit annotations could be handled as a special case of type rules (Section 4.5), but we found it more natural to separate them, as is also often done in formal expositions of type systems.

4.5 Defining type rules

A type system's rules define which operations on values of a particular type are forbidden. The Checker Framework builds in most checks of the type hierarchy. It checks that, in every assignment and pseudo-assignment, the left-hand side of the assignment is a supertype of (or the same type as) the right-hand side; for example, this assignment is not permitted:

```
 @Nullable Object myObject = ...;
  @NonNull Object myNonNullObject;
myNonNullObject = myObject;  // invalid assignment
```

The Checker Framework checks the validity of overriding and subclassing, and it prohibits inconsistent annotations at a single location.

The framework provides a base *visitor class* that performs type-checking at each node of a source file's AST. It uses the visitor design pattern to traverse Java syntax trees as provided by Sun's Tree API[2], and issues a warning whenever the type system induced by the type qualifier is violated. As with all aspects of the Checker Framework, the default behavior may be overridden, in this case by overriding methods in the framework's visitor class.

As an example, the visitor class for the Nullness type system of Section 5.3 overrides the visitor method for dereferences to issue a warning if an expression of Nullable type is

[1]`String.intern` is the only occurrence of a method that performs interning in the JDK.

[2]`http://java.sun.com/javase/6/docs/jdk/api/javac/tree/index.html`

dereferenced, as in:

```
@Nullable Object myObject = ...;
myObject.hashCode();  // invalid dereference
```

If the checker discovers violations of the type rules as it traverses the AST, it reports errors and warnings via the Java compiler's messaging mechanism [55].

As a special case, assignment can be decoupled from subtyping by overriding the `isAssignable` method, whose default implementation checks subtyping. The IGJ and Javari checkers override `isAssignable` to additionally check that fields are re-assigned only via mutable references.

4.6 Customizing the compiler interface

The Checker Framework provides a base *checker class* that is a Java annotation processor, and so serves as the entry point for the compiler plug-in.

Type system designers associate type qualifiers with a checker by writing the `@TypeQualifiers` annotation on a checker class and passing it the classes of one or more type qualifier annotations (i.e., annotations bearing `@TypeQualifier` on their declarations, as in Section 4.3) as an argument. The checker class for the Nullness checker, for instance, has the following meta-annotation on its declaration:

```
@TypeQualifiers({NonNull.class, Raw.class, Nullable.class})
```

Other annotations configure the plug-in's command-line options and the annotations that suppress its warnings. For details, see the Checker Framework manual[3].

4.7 Parametric polymorphism

The Checker Framework handles two types of polymorphism: for (qualified) types, and for qualifiers.

[3]`http://group.csail.mit.edu/pag/jsr308/current/checkers-manual.html`

4.7.1 Type polymorphism

As noted in Chapter 3, a programmer can annotate generic type arguments in a natural way, which is critical for real Java programs; the subtyping rules of Section 4.5 fully support qualified generic types.

Generic type argument inference

The base type factory performs generic type inference for the invocations of generic methods. It adheres to the Java language specification [32, §15.12.2.7] for inferring type arguments, using the most restrictive type qualifiers on type arguments when possible.

When a type variable is not used within the type of a method parameter, type arguments cannot be inferred based on the arguments of an invocation of that method (as in the JDK method `Collections.emptyList`).

If type arguments can be inferred from the arguments of an invocation of a method, type inference does not consider return types. For example, consider the following code:

```
List<String> lst = Arrays.asList("s", "t", "r");
```

under the Interning type system (Section 5.4), the return type of `Arrays.asList(...)` is `List<@Interned String>` which is not a subtype of `List<String>`[4], so the Interning checker emits a compile-time error.

Inferring the qualified types of type arguments from the actual arguments ensures type safety and is therefore preferred. In the previous example, the reference returned by `asList` could be used to mutate the array, as in the following example:

```
String[@Interned] array = ...
List<String> lst = Arrays.asList(array);
lst.set(0, new String("23"));
```

The last statement effectively inserts a non-interned value into the array.

[4]The Interning type system follows Java subtyping rules, under which `List<B>` is not a subtype of `List<A>` even if `B` is a subtype of `A`.

Generic type argument inference eliminated dozens of false positive warnings between the case studies described in Chapter 5 and an earlier version of the case studies.

Least Upper Bounds

A conditional expression ("`( ? : )`") may have true and false expressions with differing types. The overall type of the conditional expression is determined by applying capture conversion to the least upper bound the types of the tree and false expressions (according to JLS §15.25).

Determining the least upper bound of qualified types may not be possible when the expressions have the same Java types but different qualified types. For example, the least upper bound of the types `List<@NonNull String>` and `List<@Nullable String>` is `List<? extends @Nullable String super @NonNull String>`. However, Java does not allow a wildcards to have both `extends` and `super` clauses. For this reason, the true and false expressions of conditional expressions ought to have the same type, and the Checker Framework issues a warning when they are not identical.

4.7.2 Qualifier polymorphism

The Checker Framework supports type qualifier polymorphism for methods, limited to a single qualifier variable (which we have found to be adequate in practice). Thus, programmers need not introduce generics just for the sake of the qualified type system. More importantly, qualified type polymorphism (Java generics) cannot always express the most precise signature of a method.

The `@PolymorphicQualifier` meta-annotation marks an annotation as introducing qualifier polymorphism. For example, the Nullness checker includes the `@PolyNull` annotation for polymorphism over nullness. `@PolyNull` is defined as follows:

```
@PolymorphicQualifier
public @interface PolyNull { }
```

Then, a programmer can use the marked annotation as a type qualifier variable. For example, `Class.cast` returns `null` if and only if the argument is `null`:

```
@PolyNull T cast(@PolyNull Object obj)
```

For each method invocation, the Checker Framework determines the qualifier on the type of the invocation result by unifying the qualifiers of the arguments to the invocation. By default, unification chooses the least restrictive qualifier, but checkers can override this behavior as necessary.

4.8 Flow-sensitive type qualifier inference

The Checker Framework performs flow-sensitive intraprocedural qualifier inference; any checker can utilize the qualifier inference via a few lines of code. The inference may compute a more sophisticated type (that is, a subtype) for a reference than that given in its declaration. For example, the Nullness checker (Section 5.3) issues no warning for the following code:

```
@Nullable Integer jsr;
...
// valueOf signature: @NonNull Integer valueOf(String);
jsr = Integer.valueOf("308");
... jsr.toString() ... // no null dereference warning
```

because the type of `jsr` is refined to `@NonNull Integer`, from the point of its assignment to a non-null value until its next possible re-assignment. This enables a single variable to have different qualified types in different parts of its scope, and often eliminates the need for programmers to annotate method bodies, suppressing false warnings that the checker would otherwise emit.

The inference can be described as a GEN-KILL analysis. For brevity, we describe a portion of the Nullness analysis, though the framework implements it in a generic and extensible way. For the GEN portion, a reference is known to be non-null after a null check in an assert statement or a conditional, after a non-null value is assigned to it, or after a dereference (control proceeds only if the dereference succeeds, implying that the reference is non-null). For the KILL portion, a reference is no longer non-null when it may be reassigned, or when flow rejoins a branch where the reference may be null. Reassignments include assignments to possibly-aliased variables and calls to external methods where the reference is in scope.

The analysis is implemented as a visitor for Java ASTs. To compensate for redundancy in the AST, the implementation provides dataflow abstractions (e.g., the `split` and `merge` methods handle GEN-KILL sets at branches). In addition, a type system designer can specialize the analysis by extending the dataflow abstractions or, if necessary, visitor methods. The Nullness checker, for instance, extends the `scanCondition` method to account for checks against null, no matter the type of AST node that contains the condition.

Flow-sensitive inference is critical for a Nullness type system. Programmers often overload the meaning of `null` to carry additional information. This often leads to scenarios in which the nullness of a local variable (for instance) varies throughout a method. Flow-sensitive qualifier inference for nullness can refine the type of a nullable reference in which the referent is non-null.

Chapter 5

Pluggable type checkers

To demonstrate the practicality of the Checker Framework, we have written five type checkers. Section 5.1 describes the case studies that evaluate the designs and implementations of these checkers. The Basic checker (Section 5.2) applies the type rules of any type system that has no special semantics. The Nullness checker (Section 5.3) finds and verifies the absence of null pointer dereference errors. The Interning checker (Section 5.4) finds and verifies the consistent use of interning and equality testing. The Javari checker (Section 5.5) enforces reference immutability. The IGJ checker (Section 5.6) enforces reference and object immutability.

5.1 Experimental evaluation

This section summarizes case studies that evaluate our designs and implementations. Most of the case studies (approximately 400 KLOC) were completed in summer 2007, and technical reports give additional details [43], including many examples of specific errors found[1]. As one example, the author of FreePastry (`http://freepastry.rice.edu/`, 1084 files, 209 KLOC) used the Interning checker (Section 5.4) to find problems in his code.

[1]Some of our measurements differ slightly from the previous version, because the subject programs are being maintained, because of checking additional classes, and because of improvements to the checkers and framework.

Checker & Program	Size Files	Lines	ALocs	Ann.s	Errors	False pos.
Basic checker						
Checker Framework	23	6561	3376	184	0	0
Nullness checker						
Annotation file utils	49	4640	3700	107	4	5
Lookup	8	3961	1757	35	8	4
Nullness checker	58	10798	5036	167	2	45
Interning checker						
Daikon	575	224048	107776	129	9	5
Javari checker						
JOlden	48	6236	2280	451	0	0
Javari checker	7	1520	528	60	1	0
JDK (partial)	103	5478	6622	1208	0	0
IGJ checker						
JOlden	48	6236	2280	315	0	0
TinySQL	85	18159	6574	1125	0	0
Htmlparser	120	30507	11725	1386	12	4
IGJ checker	32	8691	4572	384	4	3
SVNKit	205	59221	45186	1815	13	5
Lucene	95	26828	10913	450	13	2

Table 5.1: Case study statistics. Sizes are given in files, lines, number of possible annotation locations, and number of annotations written by the programmer. Errors are runtime-reproducible problems revealed by the checker. False positives are caused by a weakness in either the type system or in the checker implementation.

Table 5.1 lists the subject programs. The annotation file utilities (distributed with the Checker Framework[2]) extract annotations from, and insert them into, source and class files. Lookup is a paragraph grep utility distributed with Daikon[3] [22], a dynamic invariant detector; the Checker Framework is described in Chapter 4; and the Nullness checker is described in Section 5.3. JOlden is a benchmark suite[4] [10]. The partial JDK is several packages from Sun's implementation[5]. The Javari checker is described in Section 5.5. TinySQL is a library implementing the SQL query language[6]. Htmlparser is a library for parsing HTML docu-

[2]`http://pag.csail.mit.edu/jsr308/`

[3]`http://pag.csail.mit.edu/daikon/`

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

[5]`http://java.sun.com/`

[6]`http://sourceforge.net/projects/tinysql/`

ments[7]. The IGJ checker is described in Section 5.6. SVNKit is a client for the Subversion revision control system[8]. Lucene is a text search engine library[9].

The sizes in Table 5.1 include libraries only if the library implementation (body) was itself annotated and type-checked. For example, each checker was analyzed along with a significant portion of the Checker Framework itself.

5.1.1 Methodology

This section presents our experimental methodology.

First, a type-system designer wrote a type checker using the Checker Framework. The designer also annotated JDK methods, by reading JDK documentation and occasionally source code.

Then, a programmer interested in preventing errors annotated a program and fixed warnings reported by the checker, until the checker issued no more warnings. In other words, the case study design is inspired by partial verification that aims to show the absence of certain problems (modulo standard static analysis caveats about reflection, unchecked libraries, suppressed warnings, etc.), rather than by bug-finding that aims to discover a few "low-hanging fruit" errors, albeit with less programmer effort. (See Section 5.3.3 for an empirical comparison of the verification and bug-finding approaches.) Therefore, the number of errors reported in Table 5.1 is less important than the fact that no others remain (modulo the guarantees of the checkers).

The programmer manually analyzed every checker warning and classified it as an error only if it could cause a problem at run time, or if an author of the code agreed that the code needed to be changed. Mere code smells count as false positives, even though refactoring would improve the code.

When fixing errors, the programmer made the smallest bug fix possible. The checkers indicated many places that the code could be refactored or its design improved (and this

[7]`http://htmlparser.sourceforge.net/`

[8]`http://svnkit.com/`

[9]`http://lucene.apache.org/java/docs/index.html`

would have also reduced the number of false positive warnings). Such changes would involve much more time and effort, and the effort would be harder to quantify, biasing the experimental results. For similar reasons, our studies analyze existing programs rather than writing new programs matched to the checker's capabilities. We note possible bias in that a few of the subject programs are the checkers themselves. The authors might have had the type system in mind while writing them, though no annotation occurred until after the checker was complete.

Warnings that cannot be eliminated via annotation, but that cannot cause incorrect user-visible behavior, count as false positives. The programmer used a `@SuppressWarnings` annotation to suppress each false positive. As an exception to this rule, when using the Nullness checker, the programmer suppressed each false positive with an assertion (e.g., "`assert x != null;`"), which had the positive side effect of checking the property at run time.

All but 6 false positives were type system weaknesses, also known as "application invariants". These manifest themselves in code that can never go wrong at run time, but for which the type system cannot prove this fact. For instance, the checker issues a false positive warning in the following code:

```
Map<String, @NonNull Integer> map;
String key;
...
if (map.containsKey(key)) {
  @NonNull Integer val = map.get(key); // false positive
}
```

`Map.get` is specified to possibly return null (if the key is not found in the map); however, in the above code the `Map.get` return value is non-null, because `key` must be in the `Map` if line 5 is reached. The other 6 false positives were caused by weaknesses in a checker implementation.

In 17 cases an edit was necessitated by a questionable coding style where, in order to create an object, a client must call both a constructor and then an additional method. The programmer inserted the helper method body, or a call to the helper method, in the

Checker	Total	Type rules	Type intro.	Flow	Compiler i/f
Basic	87	0	20	13	54
Nullness	502	44	187	258	13
Interning	209	129	61	5	13
Javari	334	55	236	n/a	43
IGJ	438	0	338	n/a	94

Table 5.2: Checker size, in non-comment non-blank lines of code. Size for integration with flow-sensitive qualifier inference is separated from the rest of the type introduction code. Qualifier definitions are omitted: they are small and have empty bodies. The compiler interface for the Nullness and Interning checkers has an empty body; `package` and `import` statements account for the majority of the code.

constructor.

Finally, JOlden and Lucene are written in pre-generics Java, so the programmer added type parameters to it before proceeding with the case study.

5.1.2 Ease of use

The Checker Framework is easy for a type system designer to use, and the resulting checker is easy for a programmer to use.

It was easy for a type system designer to write a compiler plug-in using the Checker Framework. Table 5.2 gives the sizes of the five checkers presented in this paper. Most of the methods are very short, but a few need to take advantage of the power and expressiveness of the Checker Framework. As anecdotal evidence, the Javari and IGJ checkers were written by a second-year and a third-year undergraduate, respectively. Neither was familiar with the framework, and neither had taken any classes in compilers or programming languages. As another anecdote, adding support for `@Raw` types [27] to the Nullness checker took about 1 hour. It took about 2 hours to generalize the Nullness-specific flow-sensitive type qualifier inference [43] into the framework feature of Section 4.8.

It was also easy for a programmer to use a checker. The Interning case study, and parts of the Nullness case studies, were done by programmers with no knowledge of either the framework or of the checker implementations. Subsequent feedback from external users of the checkers has confirmed their practicality. Furthermore, using a checker was quick. Almost all of the programmer's time was spent on the productive tasks of understanding and fixing errors. Annotating the program took negligible time by comparison.[10] Identifying false positives was generally easy, for three reasons: many false positives tended to stem from a small number of root causes, many of the causes were simple, and checker warnings indicate the relevant part of the code. Good integration with tools such as `javac` aided all of the tasks.

5.2 The Basic type checker for any simple type system

The Basic checker performs only checks related to the type hierarchy (Section 4.3). This is adequate for simple type systems — those with no special semantics beyond Java subtyping rules — and is ideal for prototyping.

The type system designer writes no code besides annotation definitions (which have empty bodies). The programmer names the qualifiers that make up the type system on the command line.

The Basic checker supports all of the functionality provided declaratively by the Checker Framework, including arbitrary type hierarchy DAGs, type introduction rules, qualified type and qualifier polymorphism (Section 4.7), and flow-sensitive inference (Section 4.8).

Additional examples of useful type qualifiers include `YY` (for two-digit year string), `YYYY` (for four-digit year string), which helped to detect and verify absence of Y2K errors [20]; the `localizable` qualifier to indicate where translation of user-visible messages should be performed; and qualifiers for specifying the format or encoding of a string (e.g. `XML` or `SQL`).

The Basic checker is useful for creating prototypes to experiment with these type systems and others. However, the Basic checker is limited in its abilities to enforce the invariants (e.g. checking that `YY` string literals have two-digits), and the user may need to add (qualified) type casts or wrap the source of values within some methods.

[10]However, we have since developed inference tools for the Javari and Nullness type systems. These tools, discussed in the Checker Framework manual (`http://pag.csail.mit.edu/jsr308/current/checkers-manual.html`), further reduce the annotation burden, particularly for libraries and legacy code.

5.2.1 Basic checker case study

As a case study, a type system designer used the Basic checker to define `@Fully` and `@Partly` annotations that were useful in verifying the Checker Framework itself. The framework constructs an annotated type (`AnnotatedTypeMirror`, Section 4.2) of an expression in several phases, starting from an unannotated type provided by the underlying compiler. It first adds the annotations that were explicitly written on that expression's type, then it adds resolved annotations (e.g., from type variable substitution or generic type inference), and finally it adds implicit annotations (Section 4.4); the framework must never return a partially-constructed annotated type to a checker.

The `@Fully` type qualifier indicates that construction is complete. A `@Fully` annotated type is a subtype of a `@Partly` annotated type. The programmer annotated each use of `AnnotatedTypeMirror` in the framework with `@Fully` or `@Partly` annotations to verify that the framework never returns a partially-constructed annotated type to a checker.

The case study required 55 uses of qualifier polymorphism (Section 4.7). For instance, the component type of an array type has the same annotatedness as the array type, so the programmer annotated the `getComponentType` method of the `AnnotatedArrayType` class as follows:

```
@PolyAnno AnnotatedTypeMirror getComponentType() @PolyAnno { ... }
```

5.3 The Nullness checker for null pointer errors

5.3.1 The Nullness type system

The Nullness checker implements a qualified type system in which, for every Java type `T`, `@NonNull T` is a subtype of `@Nullable T` (see Figure 5-1). As an example of the difference, a reference of type `@Nullable 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

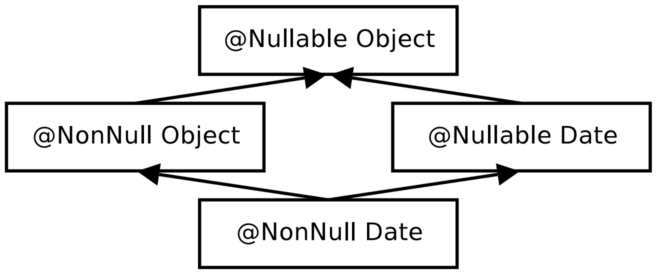

Figure 5-1: Type hierarchy for the Nullness type system. Java's `Object` is expressed as `@Nullable Object`. Programmers can omit most type qualifiers, thanks to careful choice of defaults.

pointer exception.

The Nullness checker issues a warning when an expression that does not have a `@NonNull` type (i.e. has a `@Nullable`) type is dereferenced. Additionally, as with all checkers, it warns of violations of the type system; for the Nullness checker, this is when an expression of `@NonNull` type might become null. Either problem might cause a null pointer exception at run time. The following code example illustrates both kinds of errors:

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

The Nullness checker supports the `@Raw` type qualifier for partially-initialized objects [27]. (The `@Raw` type qualifier is unrelated to the raw types of Java generics.) During the execution of a constructor, all fields of non-primitive type start out with the value null, including those with a `@NonNull` type. If the constructor calls a method, that method could dereference uninitialized `@NonNull` fields. `@Raw` prevents errors like these: if a reference has a `@Raw` type, all fields of its referent are treated as `@Nullable`. `this` implicitly has a `@Raw` type within the constructor, so it can only be passed to methods when the corresponding parameter has a `@Raw` type. Similar restrictions apply when assigning `this` to a field or invoking a method on it. Implementing support for the `@Raw` type qualifier took about an hour of work.

The Nullness checker's visitor class implements three type rules: for dereferences of possibly-null expressions ("type rules" column of Table 5.2), implicit iteration over possibly-null collections in an enhanced `for` loop, and accessing a possibly-null array. All three rules check for dereferences of a possibly-null reference; the last two account for Java's syntactic sugar. The type introduction rules add the `@NonNull` annotation to literals (except `null` gets `@Nullable`), `new` expressions, and classes used in static member accesses (e.g., `System` in `System.out`).

The Nullness checker optionally warns about a variety of other null-related coding mistakes, such as checking a value known to be (non-)null against null. These do not lead to run-time exceptions and so are not tabulated in Table 5.1, but these redundant tests and dead code are often correlated with other errors [35].

5.3.2 Type system weaknesses

Like other Nullness type systems, ours is good at stating whether a variable can be null, but not at stating the circumstances under which the variable can be null. In the Lookup program, `entry_start_re` is null if and only if `entry_stop_re` is null. After checking just one of them against null, both may be dereferenced safely. 39 of the 45 false positives in the Nullness checker case study (Figure 5.1) were due to similar complex nullness invariants, especially in AST node operations. Expressing such application invariants would require a substantially more sophisticated system, such as dependent types [46].

The flexibility of the Checker Framework permits type system designers to create more sophisticated checks even if they are not expressible in a type system. For example, the best type for `Collection.toArray` is both reflective and polymorphic, and checkers can treat it as such.

Another example from the Lookup subject program is that of a variable holding a factory method for a class. The variable is non-null if the class has no constructor and the class is not a Java `enum`; in code implementing that case, the variable is unconditionally dereferenced.

As another example, the compiler API used by the Nullness checker contains a number of methods that return null if and only if their single parameter is null. For example, the JDK method `Class.cast` has the following signature:

```
T cast(Object obj)
```

`cast` returns null if and only if `obj` is null. Using the `@PolyNull` annotation for polymorphism over nullness, the signature of `Class.cast` becomes:

```
@PolyNull T cast(@PolyNull Object obj)
```

Then, the return value of `cast` has type `@NonNull T` when `obj` is non-null.

Similarly, one variant of the `Properties.getProperty` method has the following signature:

```
String getProperty(String key, String default)
```

The return value of `getProperty` has the type `@NonNull String` if and only if the `default` parameter has the type `@NonNull String` (`Properties` does not permit mappings to null values). Using the `@PolyNull` annotation, the signature becomes

```
@PolyNull String getProperty(String key, @PolyNull String default)
```

The occurrence of invocations of methods like `Class.cast` and `Properties.getProperty` in the subject programs motivated the implementation of qualifier polymorphism in the Checker Framework.

In Table 5.1, the 4 errors detected in the "Annotation file utils" subject program is an underestimate. Before our case study, the program's author had already fully annotated the code with `@NonNull` annotations, simulated the type rules by hand, and fixed all problems that arose. After that, the programmer in the case study ran the checker, which revealed 4 additional errors.

All other subject programs used the methodology described in Section 5.1.1.

5.3.3 Errors found

The most frequent null dereference error (both in our case studies, and also in feedback from other users of the checker) resulted from failure to check a value returned by a method, especially when the method rarely returns null in practice.

A typical example from the Nullness checker subject program (simplified for presentation) is the following:

```
if (enclosing.getElement().getKind() == METHOD)
  return enclosing.getReceiverType();
```

The method `getElement()` returns null when `enclosing` represents the enclosing method of a statement in a static block. This surprised the programmer, since the Java language requires compiling the contents of a static block into a static initializer method. The warning message led the programmer to fix this error before it ever caused a null pointer exception.

As another example from the same subject program, a type warning revealed an error where the variable `nnElement` was being checked against null, but a bug fix introduced code using `nnElement` before (rather than after) the check. The programmer fixed this by reordering the statements.

Here are some other 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` (from `java.util.regex`) can return null. Checking for null permits a comprehensible error message rather than a crash.

The warnings indicated to the programmer several other problems. (Since they do not cause a null pointer exception, Table 5.1 does not count them, to avoid inflating our numbers.) As an example, the Lookup program's command-line options could set but not disable certain options: although null is used as a flag, there was nowhere that the variable could be set to null. As another example, the checker's advisories revealed dead code, due to null checks of values that cannot be null, in each of the subject programs.

We evaluated our checker against the null pointer checks of several other static analysis tools, using the Lookup subject program. Table 5.4 tabulates the results. The other tools missed all the errors, and did not indicate any locations where annotations could be added to improve their results. In their defense, they did not require annotation of the code, and their other checks (besides finding null pointer dereferences) may be more useful.

	Errors		False	Annotations
Tool	Found	Missed	warnings	written
The Checker Framework	8	0	4	35
FindBugs	0	8	1	0
JLint	0	8	8	0
PMD	0	8	0	0

Table 5.3: A comparison of our Nullness checker with other bug-finding tools.

5.3.4 Default annotation for Nullness checker

The Nullness checker treats unannotated local variables as Nullable, and all other unannotated types (including generic type arguments on local variables) as non-null. We call this default NNEL, for NonNull Except Locals. The NNEL default reduces the programmer's annotation burden, especially when combined with the flow-sensitive type inference described in Section 4.8. The default can be overridden on a class, method, or field level.

We believe the NNEL design for defaults to be novel, and our experience indicates that it is superior to other choices. NNEL combines the strengths of two previously-proposed default systems: nullable-by-default and non-null-by-default.

Nullable-by-default has the advantage of backward-compatibility, because an ordinary Java variable may always be null. However, in practice many variables have a `@NonNull` type, so this default requires many annotations.

Non-null-by-default is a syntactic convenience that does not affect the type system, but makes the source code "`@Nullable Object`" refer to the top of the checker's type hierarchy (Figure 5-1), and the source code "`Object`" refer to its non-null subtype (`@NonNull Object`). A disadvantage is that an incremental approach to annotating legacy code is not possible. However, non-null-by-default reduces clutter and annotation effort in programs that use more non-null than nullable types. Non-null types are believed to be more prevalent, so Splint, Nice, JML, and Eiffel have adopted non-null-by-default semantics [26, 27, 8, 38, 17]. (Another reason for a non-null default is to bias programmers away from using nullable variables. Every program needs some nullable variables, but they should be avoided when possible.)

	Nullable			NonNull			NNEL		
Program	Tot	Sig	Body	Tot	Sig	Body	Tot	Sig	Body
Annotation file utils	760	483	277	165	86	79	107	90	17
Lookup	382	301	81	78	31	47	35	33	2
Nullness checker		—		282	126	156	146	126	20

Table 5.4: The number of annotations required to eliminate null dereference warnings, depending on the default for nullity annotations. The total number of annotations ("Tot") is the sum of those in method/field signatures ("Sig") plus those in method/initializer bodies ("Body").

To evaluate the defaults, the programmer annotated subject programs two or three separate times, using different defaults. (Since the type system and checker are unchanged, the checker warnings indicated exactly the same errors regardless of the annotation default.) We are not aware of any previous study that quantifies the difference between using nullable-by-default and non-null-by-default, though Chalin and James [11] determined via manual inspection that about 3/4 of variables in JML code and the Eclipse compiler are dereferenced without being checked.

Table 5.4 shows our results. A non-null default requires fewer annotations than a nullable default, but NNEL is best of all. Although the nullable default is worse than the non-null default overall, it requires fewer annotations in method bodies; the flow-sensitive type inference often permitted a completely unannotated method body to type-check by inferring `@NonNull` types for some local variables. NNEL is as good as non-null for signatures, and is even better than nullable for bodies. The NNEL code was not just terser, but — more importantly — clearer to the programmers in our study. Reduced clutter directly contributes to readability. NNEL mitigates backward-compatibility issues, because programmers usually plan to annotate signatures anyway; doing so is required for modular checking and is useful and non-burdensome compared to annotating method bodies. Our choice of the NNEL default was also motivated by the observation that when using nullable-by-default, programmers most often overlooked `@NonNull` annotations on generic types; the NNEL default corrects this problem (since only the *raw* types of locals are `@Nullable` in NNEL). A potential downside of non-uniform defaults is that an unannotated type such as "`Integer`" means different things

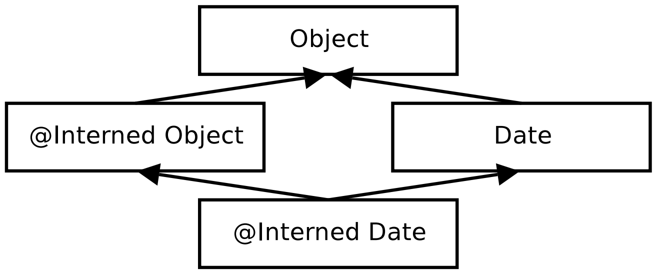

Figure 5-2: Type hierarchy for the Interning type system.

in different parts of the code. However, this was not a problem in practice, perhaps because programmers think of public declarations differently than private implementations. Further use in practice will yield more insight into the benefits of the NNEL default. We believe that the general approach embodied by the NNEL default is also applicable to other type systems.

5.4 The Interning type checker for equality-testing and interning errors

Interning, also known as canonicalizing or hash-consing, finds or creates a unique concrete representation for a given abstract value. That representation 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 representation that a client should use 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 a single, unique 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. However, misuse of interning can lead to bugs: use of `==` on distinct objects representing the same abstract value may return false, as in the expression `new Integer(22) == new Integer(22)`.

The Interning type hierarchy is depicted in Figure 5-2. We believe that ours is the first formulation of a completely backward-compatible system for interning.

5.4.1 The Interning checker

If the Interning checker issues no warnings for a given program, then all reference (in)equality tests (`==` and `!=`) in that program operate on interned types.

The visitor class (type rules) for the Interning checker has 3 parts:

1. It overrides one method to warn if either argument to a reference (in)equality operator (`==` or `!=`) is not interned. For example:

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

   In addition, since it extends the base visitor class, assignments such as the following are not permitted:

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

2. Most of the checker is code to eliminate two common sources of false positives, suppressing warnings when:

   (a) the first statement of an `equals` method is an `if` statement that returns `true` after comparing `this` and the (sole) parameter, or

   (b) the first statement of a `compareTo` method returns `0` after comparing its two parameters.

3. The checker optionally issues a warning when `equals()` is used to compare two interned objects. These warnings do not indicate a correctness problem, so Table 5.1 does not report them. However, they did enable the programmer to use `==` in several instances, making the code both clearer and faster.

The type introduction rules mark as `@Interned`: string and class literals, values of primitive, enum, or `Class` type, and the result of the `String.intern` method. The Interning checker requires no library annotations, since the only library method that affects interning is `String.intern`.

The implementation of the Interning checker is provided in Appendix A.

5.4.2 Interning case study

We evaluated the Interning checker by applying it to Daikon [22]. Daikon is a dynamic invariant detector — that is, it observes program executions and generalizes from observed values to likely invariants. Not counting third-party libraries, (even those included in source form in the Daikon distribution), Daikon consists of approximately 250 KLOC of Java code. Daikon is relatively mature, at least by the standards of research software. The Daikon tool has been used in about 100 publications, as well as many additional uses (e.g., by working developers) that did not result in a published paper.

Daikon is a good subject program because memory usage is the limiting factor in its scalability [45]. Daikon uses the interning design pattern extensively. 1170 lines of comments or code contain “canonical”, “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()`), and 137 others can be viewed as checking both interning and other types of data consistency (e.g., `x.ppt==y.ppt`). The programmer annotated 11 files (12 KLOC) in Daikon with 127 `@Interned` annotations; these files contain more than half of Daikon’s 1170 interning comments/calls. The distribution of interning is uneven in the code: 72% of the files have no interning comments/calls, and 87% have no more than 2. Furthermore, manual spot inspection indicates that these files do not use interning in error-prone ways. 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 array types. The Daikon developers use an Emacs plug-in that checks code for `String`-related interning errors whenever a file is saved.

Despite the fact that its programmers have spent considerable time and effort validating its use of interning, annotating only part of Daikon revealed 9 errors and 2 optimization opportunities.

The programmer performed annotation and bug fixing (see below), but no refactoring nor algorithmic changes.

5.4.3 Errors found

The Interning checker revealed 9 previously unknown interning-related errors in Daikon, 2 performance bugs (unnecessary interning), and a design flaw. The programmer 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. The programmer added 2 missing calls to intern methods in `read_data`, so its result is always interned.

A code comment indicated that the `VarInfo.str_name` field was interned, but `VarInfo` constructors failed to intern it on 5 occasions — almost half of all locations where the field is set in 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. The `simplify_expression` method performs algebraic simplification by side effect (side effects are 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 was 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. The programmer 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. `VarInfoName` represents variable names, their formatting, and their relationships to one another and to program points. All external references to this class are interned (and the programmer verified manually that all clients treat them properly), but within the class body instances are sometimes uninterned (for instance, in the middle of a sequence of operations within a method). The programmer discovered locations where uninterned instances could leak to the outside as private fields or as subcomponents of interned references, but was unable 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 the 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 Interning type checker is easy to use and can be extremely fruitful in identifying errors.

5.4.4 False positives

The programmer added 9 `@SuppressWarnings` annotations to eliminate false positives, as tabulated in Table 5.1. The false positives in Table 5.1 are due to casts in `intern` methods, tests in equality methods, and an application invariant: checking whether a variable is still set to its interned initial value can safely use `==`, even if the variable's type is not interned in general.

Beyond the 5 `@SuppressWarnings` annotations noted in Table 5.1, the programmer added 4 additional ones, to account for calls to files that the programmer did not annotate.

5.5 The Javari checker for mutability errors

5.5.1 The Javari type system

A mutation error occurs when a side effect modifies the state of an object that should not be changed. Mutation errors are difficult to detect: the object is often (correctly) mutated

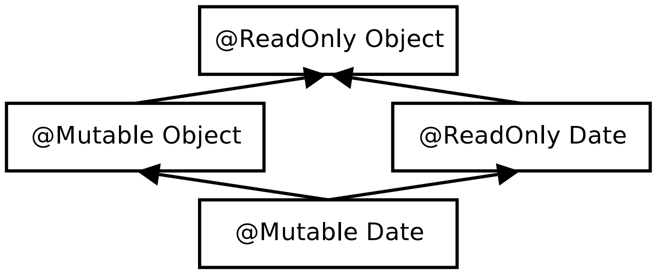

Figure 5-3: Type hierarchy for Javari's ReadOnly type qualifier.

in other parts of the code, and a mutation error is not immediately detected at run time. The Javari [6, 52] type system enables compile-time detection and prevention of mutation errors.

Javari is an extension of the Java language that permits the specification and compile-time verification of immutability constraints. Figure 5-3 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 read-only and mutable types.
- The `@PolyRead` 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 `@PolyRead` to indicate that its parameter and return value are either both read-only or both non-read-only. `@PolyRead` was previously known as `@RoMaybe`.

The type system is specified in greater detail elsewhere [52], including why it is necessary and correct for `@ReadOnly` types to be supertypes of their unqualified counterparts.

5.5.2 The Javari checker

The visitor class for the Javari checker overrides each method that handles an operation with the potential to perform a side effect — notably field assignment — in order to warn if mutation occurs on a reference with a read-only type:

```
String localString;

// mutable method
void aMutableMethod() {
   localString = "a";      // no error
}

// readonly method
void anotherMethod() @ReadOnly {
   localString = "a";      // error
   aMutableMethod();       // error
}
```

The type introduction rules handle features that make the type of a reference dependent on the context, including field mutability inherited from the current reference (Javari's "this-mutable") and parametricity over mutability including wildcards (`@PolyRead`). The following code fragment demonstrates the use of `@PolyRead`:

```
class DWrapper {
  Date localDate;

  DWrapper(@PolyRead Date d) @PolyRead {
    // new object has same mutability
    // as constructor parameter
    localDate = d;
  }
}

@ReadOnly Date roDate;
Date mutDate;
```

```
...

DWrapper w1 = new DWrapper(roDate);
// error: cannot assign readonly to mutable
// other assignments are legal:
DWrapper w2 = new DWrapper(mutDate);
@ReadOnly DWrapper w3 = new DWrapper(roDate);
@ReadOnly DWrapper w4 = new DWrapper(mutDate);
```

5.5.3 Errors found

The Javari checker found a mutability bug in its own implementation. A global variable containing information about the state of the checker was mutated when the checker visited the AST node for an inner class, but was not reset upon exiting. (The programmer's 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 programmer found annotating his own code to be easy and fast. The most difficult part of the case study was annotating largely undocumented third-party code. Quite a few methods modified their formal parameters, but this important and often surprising fact was never documented in the code the programmer examined.

The most difficult method to annotate was `Collection.toArray` method, which has the following signature:

```
<T> T[] toArray(T[] a)
```

In addition to reflective complexities noted earlier, `toArray` modifies its argument exactly if the argument has greater size than the receiver (i.e. the size of the array argument is greater than the size of the collection on which `toArray` was invoked).

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

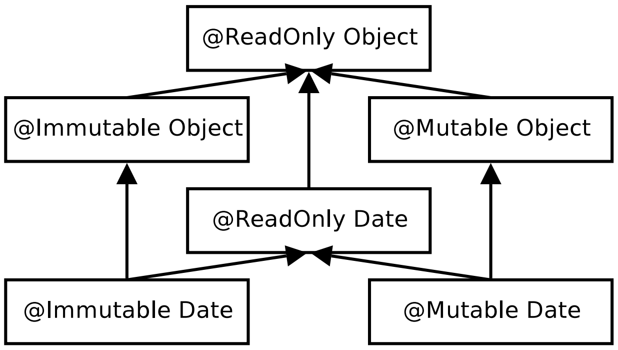

Figure 5-4: Type hierarchy for three of IGJ's type qualifiers.

explicit; the flow-sensitive analysis described in Section 4.8 would have eliminated the need for these.

The programmer was able to annotate more local variables in the Javari checker than in the JOlden benchmark, due to better encapsulation and greater incidence of getter methods. Most of the annotations were `@ReadOnly` (288 annotations on classes, 514 annotations on libraries and interfaces). The programmer never used the `@QReadOnly` annotation; the default inherited mutability was expressive enough. The programmer used `@PolyRead` extensively: on almost every getter method and most constructors, but nowhere else. The programmer 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. The programmer used `@Assignable` 16 times, all while annotating a set of inner anonymous classes in JOlden that extended `Enumeration`, that could conceivably be read-only, and that required a reference to the last visited item to be assignable.

5.6 The IGJ checker for mutability errors

5.6.1 The IGJ type system

Immutability Generic Java (IGJ) [57] is a Java language extension that expresses immutability constraints. Like the Javari language described in Section 5.5, it is motivated by the fact that a compiler-checked immutability guarantee detects and prevents errors, provides useful documentation, facilitates reasoning, and enables optimizations. However, the two type systems are quite different. 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 object can never be mutated).

The IGJ type system ensures that no object can be mutated through a read-only reference. The following code illustrates type errors:

```
@Immutable Date myImmutableDate = ...;
@ReadOnly Date myReadOnlyDate = ...;
myImmutableDate = myMutableDate; // invalid assignment
myImmutableDate.setMonth(2);     // invalid invocation
myReadOnlyDate.setMonth(2);      // invalid invocation
```

IGJ is the first proposal for enforcing object immutability within Java's syntax and type system, and its reference immutability is more expressive than previous work. IGJ also permits covariant changes of type parameters in a type-safe manner, e.g., `@ReadOnly List<Integer>` is a subtype of a `@ReadOnly List<Number>`.

Every reference is annotated as `@Immutable`, `@ReadOnly`, `@Mutable` (the default), or `@AssignsFields`; Figure 5-4 illustrates the relationship among the first three of these, and the following list summarizes their semantics:

- A reference with an `@Immutable` type refers to an immutable object, which cannot be mutated via the immutable reference or any aliasing reference.

- A reference with a `@ReadOnly` type provides only read-only access to its referent. No mutation may occur via the reference, but mutation of the referent is possible via an aliasing reference.

- A reference with a `@Mutable` type refers to an 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 of the enclosing object and may be reassigned, irrespective of the immutability of the enclosing object.

- 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 IGJ checker does not extend the visitor class; the base functionality provided by the framework is sufficient for the IGJ checker.

The checker class overrides the handler for re-assignment to warn against re-assignment of non-assignable (i.e. `final`) fields. It inherits functionality to warn against invoking a mutating method on a read-only reference and about assignment of incompatible immutability types (e.g., assigning an immutable object to a mutable reference). The type introduction rules handle context-sensitive references, including parametricity over mutability including wildcards (`@I`); resolve mutabilities not explicitly written on the code (i.e., inherited from a parent reference, determined with the mutability wildcard `@I`, or specified by default annotation); add `@Immutable` to literals (except `null`) and primitive types; and infer the immutability types for method return values.

The checker also emits an optional warning when casts increase the mutability access of a reference.

5.6.2 Errors found

The IGJ checker revealed 42 representation exposure errors in the case study programs. For example, in the `SVNKit` library, the SSH authentication class constructor takes the private key as a `char` array, and assigns it to a private field without copying; an accessor method also returns that private field without copying. Clients of either method can freely mutate the array's contents. In Apache's Lucene, `Document` stores a list of `Fieldable` items it contains. `Document.getFields` returns a mutable raw reference to the list. Clients of that method can mutate the list contents and add non-`Fieldable` items.

```
public static
QuadTreeNode createTree(QuadTreeNode parent,...) {
  QuadTreeNode node;
  if (...) { node = new BlackNode(...); }
  else if (...) { node = new WhiteNode(...); }
  else {
    node = new GreyNode(...);
    sw = createTree(node, ...);
    se=...; nw=...; ne=...;
    node.setChildren(sw,se,nw,ne);
  }
  return node;
}
```

Figure 5-5: The `QuadTreeNode.createTree` method of the `perimeter` program. Class `QuadTreeNode` should be immutable, so the call to `setChildren` on line 10 fails to type-check.

The checker also revealed an error in the Checker Framework itself. `AnnotatedTypeFactory` cached the resulting mutable `AnnotatedTypeMirror` for `Elements` without copying. The `AnnotatedTypeMirror` were mutated afterwards, corrupting the result for subsequent retrieval of the `Element`.

The `perimeter` program from the JOlden benchmark computes the perimeter of a region in a binary image represented by a quad-tree. This program has ten classes in three hierarchies. All instances of `Quadrant` and `QuadTreeNode` are immutable. Therefore, the programmer transformed these two classes into immutable classes, which turned all their 7 subclasses into immutable classes as well.

Conversion to IGJ also allowed the programmer to find and fix a conceptual problem in several immutable classes. For example, in JOlden's `QuadTreeNode` (see Figure 5-5) the constructor left the object in an inconsistent state that was later corrected by another method. This is illegal when a class is immutable; the second method is not permitted to modify the (immutable) object. The programmer solved such problems by adding parameters to the constructor and factory method to grant access to the complete state of the new object, or by moving all of the logic of object construction into a single method.

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 mutates that object according to the argument to `getSlotsTable`. `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.

Preliminary conversion to IGJ revealed a code smell related to a complex immutability specification in Google Guice[11], a dependency injection framework for Java. `InjectorImpl.findBindingsByType` returns a read-only list of the bindings for a type. If the type has a binding, the returned list is backed by an internal map, so future changes to the bindings are reflected in the list, except for the removal of the type in the binding. If the type has no binding, it returns an empty immutable list that does not reflect future biding changes.

Pre-existing unchecked casts (due to Java generics limitations) in the subject programs led to 11 false positives. The other 3 false positives stemmed from AST visitors. In the IGJ checker and framework, visitors are used to collect data on types (via read-only references) and to add implicit annotations on the types (via mutable references). To eliminate these false positives, the programmer could write separate abstract visitor classes: one for read-only nodes and one for mutable notes. Using the design pattern, the class for each node in the structure has an `accept` method, typically implemented as follows:

```
public <R> R accept(Visitor<R> visitor) {
  return visitor.visit(this);
}
```

where `R` is the return type of visitor method for that node. If the `accept` method's receiver has a `@ReadOnly` type, the visitor is passed a read-only reference to the object, and cannot mutate it; in another words, we cannot have a mutating visitor. On the other hand, if the `accept` method's receiver is mutable, the `accept` method cannot be invoked via a read-only reference, even if the visitor does not mutate the object.

11 false positives were due to unchecked casts. The IGJ type system inherits the Java 5

[11]`http://code.google.com/p/google-guice/`

limitation of unchecked casts for generic types. For example, In `SVNReader.getMap(Object[], int)`, an `Object` was casted to `Map<?,?>`. This cast cannot be checked soundly and may allow for a mutable reference to immutable objects. The IGJ checker is only sound when used to check type-safe Java code.

Adding annotations made the code easier to understand because the annotations provide clear, concise documentation, especially for method return types. For example, they distinguished among unmodifiable and modifiable collections.

The annotated IGJ programs use both immutable *classes* and immutable *objects*. Every object of an immutable class is immutable, but greater flexibility is achieved by the ability to specify particular objects of other classes as immutable. The annotated SVNKit program uses immutable objects for `Date` objects that represent the creation and expiration times for file locks, the URL to the repository (using IGJ, a programmer 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. The programmer noted other places that code refactoring would permit the use of immutable objects where immutable classes are currently used, increasing flexibility.

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

We gained insight into how IGJ's type rules apply in practice. Less than 10% of classes had constructors that call setter methods. Programmers showed discipline regarding the mutability of references: no single variable was used both to mutate mutable references and to refer to read-only references. Most fields re-used the containing class's mutability. The programmer used few mutable fields; one of the rare exceptions was a collection (in `SVNErrorCode`) that contains all `SVNErrorCodes` ever created. The programmer used `@Assignable` fields only 13 times, to mark as `@ReadOnly` the receiver of: a tree rebalancing operation; a method that resizes a buffer without mutating the contents; and getter methods that lazily initialize their fields.

Chapter 6

Related work

6.1 Frameworks

The idea of pluggable types is not new, but ours is the first practical framework for, and evaluation of, pluggable type systems in a mainstream object-oriented language. Several previous attempts suggest that this is an important goal, but not simply a matter of engineering.

Several previous attempts have been made to build a framework for pluggable type systems in Java. The most direct comparison comes from the fact that JQual [34], JavaCOP [3], and our framework have all been used to implement the Javari [52] type system for enforcing reference immutability. The version implemented in our framework supports the entire Javari language (5 keywords). The JQual and JavaCOP versions have only partial support for 1 keyword (`readonly`), and neither one correctly implements method overriding, a key feature of an object-oriented language. The JavaCOP version has never been run on a real program; the JQual one has but is neither scalable nor sound [4]. Another point of comparison is JavaCOP's Nullness type system. Initially ineffective on real programs, recent work [41] has enabled it to scale to programs as large as 948 LOC, albeit with higher false positive rates than our Nullness checker. The JavaCOP Nullness checker, at 418 non-comment, non-blank lines, is smaller than ours (502 lines), but lacks functionality present in ours such as support for generics, arrays, and fields, checking implicit dereferences in foreach loops and array accesses, customizable default annotations, the `@Nullable` annotation, optional warnings about redundant checks and casts, optional warning suppression, other command-line options, etc.

Both JQual and JavaCOP support a declarative syntax for type system rules. This is higher-level but less expressive than the Checker Framework, which uses declarative syntax only for the type qualifier hierarchy and the qualifier introduction rules. This reflects a difference in design philosophy: they created their rule syntax first, whereas we first focused on practicality and expressiveness, introducing declarative syntax only after multiple case studies made a compelling case.

A declarative syntax even for type rules would have a number of benefits. Research papers define type systems in a syntax-directed manner; a similar implementation may be more readable and less error-prone. However, many research papers define their own conceptual framework and rule formalism, so a general implementation framework might not be applicable to new and expressive type systems. For example, JQual handles only a very restricted variety of type qualifier. To implement a type system in JavaCOP requires writing both JavaCOP-specific declarative pattern-matching rules, and also procedural Java helper code [3]; the declarative and procedural parts are not integrated as in our system. Another advantage of a declarative syntax is the potential to verify the implementation of the rules. However, any end-to-end guarantee about the tool that programmers use requires verifying Java helper code and the framework itself. So far, we have not found type rules to be particularly verbose or difficult to express in our framework, nor have the type rules been a significant source of bugs. It would be interesting to compare the difficulty of writing checkers, such as the one for Javari, in multiple frameworks, but first all the frameworks must be capable of creating the checkers. Future work should address the challenge of creating a syntax framework that permits purely declarative specification (and possibly verification) of expressive type systems for which the framework was not specifically designed. It would also be interesting to use a proposed declarative syntax as a front end to a robust framework such as the Checker Framework, permitting realistic evaluation of the syntax.

JavaCOP was first released[1] after we completed our case studies [43]. Markstrum [41] reports that the JavaCOP framework has recently acquired some of the features of the Checker Framework, such as flow-sensitive type inference[2] and integration with `javac`[3]. As of May 2008 these features do not seem to be a documented part of the JavaCOP release.

In some respects, the JavaCOP framework provides less functionality than the Checker Framework. For example, it does not construct qualified types (checker writers are on their own with generics). JavaCOP had a declarative syntax earlier than the Checker Framework, though the designs are rather different. Programmers using JavaCOP checkers suffer from the inadequacies of Java 5 annotations, limiting expressiveness and causing unnecessary false positives. The JavaCOP authors have been unable to run JavaCOP's type checkers on substantial programs. A strength of JavaCOP is its pattern-matching syntax that concisely expresses style checkers (e.g., "any visitor class with a field of type `Node` must override `visitChildren`"), and case studies [41] suggest that may be the context in which JavaCOP really shines.

Fong [30] 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.

6.2 Inference

JQual [34] supports the addition of user-defined type qualifiers to Java. JQual differs from our work in two key ways.

First, JQual performs type inference rather than type checking. Since type inference is a harder problem than type checking, JQual's restriction to simpler type rules is under-

[1]`http://www.cs.ucla.edu/~smarkstr/javacop/`

[2]Publicly available and documented in our framework, and downloaded and examined by the JavaCOP authors, in February 2008.

[3]JavaCOP uses the private javac internal AST rather than the documented Tree API as the Checker Framework does. Another difference is that JavaCOP adds a new pass rather than integrating with the standard annotation processor switch.

standable. JQual can be seen as translating the ideas of earlier CQual [31] 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.

Second, JQual is less expressive, with a focus on type systems containing a single type qualifier that induces either a supertype or a subtype of the unqualified type. JQual does not handle Java generics—it has an incompatible notion of parametric polymorphism and it changes Java's overriding rules. JQual is not scalable [34, 4], so an experimental comparison is impossible. Our framework requires annotations on signatures, which has benefits in terms of documentation. Given signature annotations, our framework's local qualifier inference seems to be as effective as a more complex full type inference would be. Our framework interfaces with scalable inference tools for the Nullness and Javari type systems.

By contrast, we target richer type qualifier systems and also Java's full built-in type system. 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 `@PolyRead` 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 fragment of Javari (Section 5.5).

Our NNEL (NonNull Except Locals) approach can be viewed as being similar to type inference: users can leave bodies largely unannotated. Even in the presence of type inference, it is still useful to annotate interfaces: as documentation, for modular checking, or due to limitations of type inference.

6.3 Null pointer dereference checking

Null pointer errors are a bugaboo of programmers, and significant effort has been devoted to tools that can eradicate them. Engelen [21] ran a significant number of null-checking tools and reports on their strengths and weaknesses; Chalin and James [11] give another recent survey. We mention four notable practical tools. ESC/Java [29] 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 [16]. This is more powerful than a type system, but suffers from scalability limitations. The JastAdd extensible Java compiler [18] includes a module for checking and inferencing of non-null types [19] (and JastAdd could theoretically be used as a framework to build other type systems). To handle manipulation of partially-initialized objects, JastAdd implements a raw type system [27], which increases the number of safe dereferences in the program from 69% to 71%. JastAdd has not yet been extended to full generics and other features of Java. The JACK Java Annotation ChecKer [39] is similar to JastAdd and the Checker Framework in that all use flow-sensitivity and a raw type system and have been applied to nontrivial programs. Unlike JastAdd but like the Checker Framework, JACK is a checker rather than an inference system. The null pointer bug module of FindBugs [35] takes a very different approach than 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.

6.4 Interning

Interning (use of a canonical representation) has been used since at least the 1950s; Ershov [24] discusses checking for duplicate formulas in an arithmetic optimizer. Interning has been widely used in Lisp data structures [33, 2], where the name "hash-consing" referred to the *cons*truction of objects making use of a *hash* table. More recently, Vaziri et al. [53] give a declarative syntax for specifying the interning pattern in Java. They use the term "relation type" for an interned class. They found equality-checking and hash code bugs similar to ours. Marinov and O'Callahan's [40] dynamic analysis 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 [56]. We are not aware of a previous implementation as a type qualifier. As a result, our system is more flexible, and less disruptive to use, than previous interning approaches [40, 28, 53] in that it neither requires all objects of a given type to be interned nor gives interned objects a different Java type than uninterned ones.

6.5 Javari

Our implementation is the first checker for the complete Javari language [52, 51], and incorporates several improvements that are described in a technical report. There have been three previous attempts to implement Javari. Birka [6] implemented, via directly modifying the Jikes compiler, a syntactic variant of the Javari2004 language, an early design that conflates assignability with mutability and lacks support for generics, among other differences from Javari. Birka's case studies involved 160,000 lines of annotated code. The JavaCOP [3] and JQual [34] frameworks 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 are essential for practical use. JavaCOP's fragmentary implementation was never executed on a real program. JQual has been evaluated, and the JQual inference results were accurate for 35 out of the 50 variables that the authors examined by hand. This comparison illustrates the Checker Framework's greater expressiveness and usability.

Javarifier [51, 13, 48] is a sound and precise type inference for Javari. Artzi et al. [4] give a detailed comparison of four immutability inference tools, including JQual and Javarifier.

6.6 IGJ

Our implementation is the second checker for the IGJ language. The previous IGJ dialect [57] did not permit the (im)mutability of array elements to be specified. The previous dialect permitted some sound subtyping that is illegal in Java (and thus is forbidden by our new checker), such as `@ReadOnly List<Integer>` $\subseteq$ `@ReadOnly List<Object>`.

6.7 Type qualifier systems

Additional examples of useful type qualifiers abound. We mention just a few others. Java uses `final` to indicate a reference may not be assigned to (this is orthogonal to the notion of immutability of the referred-to object). C uses the `const`, `volatile`, and `restrict` type qualifiers.

Type qualifiers `YY` for two-digit year strings and `YYYY` for four-digit year strings helped to detect, then verify the absence of, Y2K errors [20].

Range constraints, also known as ranged types, can indicate that a particular `int` has a value between 0 and 10; these are often desirable in realtime code and in other applications, and are supported in languages such as Ada and Pascal.

Type qualifiers can indicate data that originated from an untrustworthy source [42, 54]; examples for C include `user` vs. `kernel` indicating user-space and kernel-space pointers in order to prevent attacks on operating systems [36], and `tainted` for strings that originated in user input and that should not be used as a format string [49].

A `localizable` qualifier can indicate where translation of user-visible messages should be performed. Annotations can indicate other properties of its contents, such as the format (e.g., XML, SQL, human language, etc.) or encoding of a string (multibyte, UTF, etc.). An `interned` qualifier can indicate which objects have been converted to canonical form and thus may be compared via object equality. Type qualifiers such as `unique` and `unaliased` can express properties about pointers and aliases [25, 12]; other qualifiers can detect and prevent deadlock in concurrent programs [31, 1]. Flow-sensitive type qualifiers [31] can express typestate properties such as whether a file is in the open, read, write, read/write, or closed state, and can guarantee that a file is opened for reading before it is read, etc. The Vault language's type guards and capability states are similar [15].

Chapter 7

Conclusion

We conclude with a discussion of possibilities for future work (Section 7.1), a summary of the contributions of this research (Section 7.2), and a collection of lessons learned (Section 7.3).

7.1 Future work

The construction of a framework for building pluggable type checkers permits both a variety of new type checkers and several extensions for the framework itself. The following is a list of opportunities for future work.

7.1.1 Type checkers

- Extensions to the Basic checker (Section 5.2) to expose a richer set of the framework's functionality. Currently, the Basic checker only permits a user to specify a set of type qualifiers and (using the declarative syntax in Section 4.4) some of the properties of these qualifiers. Interactions with other checkers, checker-specific error messages, and specialization of the flow-sensitive inference cannot be specified. One potential solution involves exposing this functionality through additional declarative syntax.

- New type checkers for a variety of type qualifier systems. Among those proposed are checkers for concurrency and thread-safety, numeric sign errors (e.g., `@Positive` and `@NonNegative`), data tainting, and string representation (e.g., `@XML`, `@RegExp`, `@SQL`).

7.1.2 The Checker Framework

- More powerful inference than the flow-sensitive intraprocedural analysis described in Section 4.8. In particular, an interprocedural analysis could help address some of the shortcomings of the intraprocedural analysis when dealing with fields by determining whether a method invocation may modify a field.

- Declarative syntax beyond that described in Sections 4.3 and 4.4. Declarative syntax reduces the amount of code required to create a checker and simplifies the checker implementations, but the framework does not currently provide a declarative way to specify type rules, advanced type introductions, and specializations to flow-sensitive type inference.

- Interactions between type systems. Specifying the relationships between different type qualifiers in different type system extensions could improve various analyses and help programmers find more bugs in their programs. For instance, if an object is interned it should also be immutable. A programmer simultaneously using both the Interning (Section 5.4) and IGJ (Section 5.6) checkers could receive a warning whenever an object violates this constraint.

  As a second example, the flow-sensitive intraprocedural qualifier inference assumes that properties of fields may no longer hold after a method call, since the method may modify those fields. However, if a method's receiver has a `@ReadOnly` annotation (from Javari or IGJ), it cannot modify its fields, so the inference does not need to clear inferred annotations for fields after calls to that method.

- Typestate checking. As an example, one may specify `@Open` and `@Closed` as typestate qualifiers for the type `File`, and use them to ensure that the `read` method is only called on references of type `@Open File`. The current implementation of flow-sensitive inference may be sufficient for some typestate checkers, but explicit support could guarantee applicability for typestate checkers in general.

7.2 Summary of contributions

The Checker Framework is an expressive, easy-to-use, and effective system for defining pluggable type systems for Java. It provides declarative and procedural mechanisms for expressing type systems, an expressive programming language syntax for programmers, and integration with standard APIs and tools. Our case studies shed light not only on the positive qualities of the Checker Framework, but also on the type systems themselves.

The contributions of this research include the following.

- A backward-compatible syntax for writing qualified types that extends the Java language annotation system. The extension is naturally integrated with the Java language, and annotations are represented in the class file format. The system, now known by its Sun codename "JSR 308", is planned for inclusion in the Java 7 language.

- The Checker Framework for expressing the type rules that are enforced by a checker — a type-checking compiler plug-in. The framework makes simple type systems easy to implement, and is expressive enough that powerful type systems are possible to implement. The framework provides a representation of annotated types. It offers declarative syntax for many common tasks in defining a type system, including declaring the type hierarchy, specifying type introduction rules, type and qualifier polymorphism, and flow-sensitive local type qualifier inference. For comprehensibility, portability, and robustness, the framework is integrated with standard Java tools and APIs.

- Five checkers written using the Checker Framework. The Basic checker permits use of any type qualifier, with no type rules beyond standard Java subtyping rules. The Nullness checker verifies the absence of null pointer dereference errors. The Interning checker verifies the consistent use of interning and equality testing. The Javari checker enforces reference immutability. The IGJ checker enforces reference and object immutability. The checkers are of value in their own right, to help programmers to detect and prevent errors. Construction of these checkers also indicates the ease of using the framework and the usability of the resulting checker.

- A new approach to finding equality errors that is based purely on a type system and is fully backward-compatible.

- An empirical evaluation of the previous proposals for defaults in a Nullness type system. This led us to a new default proposal, named NNEL (NonNull Except Locals), that significantly reduces the annotation burden. Together with flow-sensitive type inference, it nearly eliminates annotations within method bodies.

- Significant case studies of running the checkers on real programs. The checkers scale to programs of >200 KLOC, and they revealed bugs in every codebase to which we applied them. Annotation of the programs indicates that our syntax proposals maintain the feel of Java. Use of the checkers indicates that the framework yields scalable tools that integrate well with developers' practice and environments. The tools are effective at finding bugs or proving their absence. They have a relatively low annotation burden and manageable false positive rates.

- New insights about previously-known type systems (see Section 7.3).

- Public releases, with source code and substantial documentation, of the JSR 308 extended annotations Java compiler, the Checker Framework, and the checkers, at `http://pag.csail.mit.edu/jsr308/`. (The first public release was in January 2007.) Additional details can be found in the Checker Framework documentation. We hope that programmers will use the tools to improve their programs, and that type theorists will use them to realistically evaluate their type system proposals.

7.3 Lessons learned

To date, it has been very difficult to evaluate a type system in practice, which requires writing a robust, scalable custom compiler that extends an industrial-strength programming language. As a result, too many type systems have been proposed without being realistically evaluated. Our work was motivated by the desire to enable researchers to more easily and effectively evaluate their proposals. Although three of the type systems we implemented have seen significant experimentation (160 KLOC in an earlier version of Javari [6], 106 KLOC in IGJ [57], many implementations of Nullness), nonetheless our more realistic implementation yielded new insights into both the type systems and into tools for building type checkers. We now note some of these lessons learned.

7.3.1 Javari

A previous formalization of Javari required formal parameter types to be covariant, not for reasons of type soundness but because it simplified the exposition and a correctness proof. We found this restriction (also present by default in the JastAdd framework) unworkable in practice and lifted it in our implementation. We discovered an ambiguous inference rule in a previous formalism; while not strictly incorrect, it was subject to misinterpretation. We discovered and corrected a problem with inference of polymorphic type parameters. And, we made the treatment of fields more precise.

7.3.2 IGJ

A rich immutability type system is advantageous; many programs used class, object, and reference immutability in different parts or for different classes. The case studies revealed some new limitations of the IGJ type system: it does not adequately support the visitor design pattern or callback methods.

In just two cases, the programmer would have liked multiple immutability parameters for an object. The return value of `Map.keySet` allows removal but disallows insertion. The return value of `Arrays.asList` is a mutable list with a fixed size; it allows changing elements but not insertion nor removal.

IGJ was inspired by Java's generics system. To our surprise, the programmer preferred annotation syntax to the original IGJ dialect. The original IGJ dialect mixes immutability and generic arguments in the same type parameters list, as in `List<Mutable, Date<ReadOnly>>`. Prefix modifiers such as `@Mutable List<@ReadOnly Date>` felt more natural to the programmer.

7.3.3 Nullness

Nullness checkers are among the best-studied static analyses. Nonetheless, our work reveals some new insights. Observing programmers and programs led us to the NonNull Except Locals (NNEL) default, which significantly reduces the user annotation burden and serves as a surrogate for local type inference. The idea is generalizable to other type qualifiers besides `@NonNull`.

Another observation is that a Nullness checker is not necessarily a good example for generalizing to other type systems. Many application invariants involve nullness, because programmers imbue `null` with considerable run-time-checkable semantic meaning. For instance, it can indicate uninitialized variables, option types, and other special cases. Compared to other type systems, programmers must suppress relatively more false positives with null checks, and flow sensitivity is an absolute must. Flow sensitivity offers more modest benefits in other type systems, apparently thanks to programmers' more disciplined use of those types. For example, the inference only reduced the number of necessary annotations in the Interning case study by only 3.

7.3.4 Expressive annotations

The ability to annotate generic types makes a qualitative difference in the usability of a checker. The same is true for arrays: while some people expect them to be rare in Java code, they are pervasive in practice. Lack of support for array annotations was the biggest problem with an earlier IGJ implementation [57], and in our case studies, annotating arrays revealed new errors compared to that implementation.

Some developers are skeptical of the need for receiver annotations, but they are distinct from both method and return value annotations. Our case studies demonstrate that they are needed in every type system we considered. Even the Nullness checker uses them, for `@Raw` annotations, although each receiver is known to be non-null.

7.3.5 Polymorphism

Our case studies confirm that qualifier polymorphism and type polymorphism are complementary: neither one subsumes the other, and both are required for a practical type system. Qualifier polymorphism expresses context sensitivity in ways Java generics cannot, and avoids the need to rewrite code even when generics suffice. Qualifier polymorphism is built into Javari and IGJ, but after we found it necessary in the Nullness checker, and useful in the Interning and Basic checkers, we promoted it to the framework. Given support for Java generics, we found polymorphism over a single qualifier variable to be sufficient; there was no real need for multiple qualifier variables, much less for subtype constraints among them.

Supporting Java generics dominated every other problem in the framework design and implementation and in the design of the type systems. While it may be more expedient to ignore generics and focus on the core of the type system, or to formalize a variant of Java generics, those strategies run the risk of irrelevancy in practice. Further experimentation may lead us to promote more features of specific type systems, such as Javari's extension of Java wildcards, into the framework.

7.3.6 Framework design

Our framework differs from some other designs in that type system designers code some or all of their type rules in Java. The rules tend to be short and readable without sacrificing expressiveness. Our design is vindicated by the ability to create type checkers, such as that of Javari, that the authors of other frameworks tried but failed to write. Several of our checkers required sophisticated processing that no framework known to us directly supports. It is impractical to build support for every future type system into a framework. Even for relatively simple type systems, special cases, complex library methods, and heuristics make the power of procedural abstraction welcome. We conclude that the checker and the compiler should be integrated but decoupled.

Use of an expressive framework has other advantages besides type checking. For example, we wrote a specialized “checker” for testing purposes. It compares an expression’s annotated type to an expected type that is written in an adjacent stylized comment in the same Java source file.

One important design decision was the interface to AST trees and symbol types. An earlier version of our framework essentially used a pair consisting of the unannotated type (as provided by the compiler) and the set of annotation locations within the type. Changing the representation eliminated much complexity and many bugs, especially for our support of generic types.

7.3.7 Inference

Inference of type annotations has the potential to greatly reduce the programmer’s annotation burden. However, inference is not always necessary, particularly when a programmer adds annotations to replace existing comments, or when the programmer focuses attention on only part of a program. Inference is much more important for libraries when the default qualifier is not the root of the type hierarchy (e.g., Javari and IGJ). Existing inference tools tend to scale poorly. After iterating for many months offering bug reports on a JastAdd static non-null inference tool (which was always more practical than other non-null inference systems, has since become solid, and now supports the JSR 308 annotation syntax), we wrote our own sound, dynamic nullable inference tool in a weekend. Just as the Checker Framework filled a need for type checking, there is a need for robust, scalable, expressive type frameworks that specifically support static inference.

7.3.8 Complexity of simple type systems

Simple qualified type systems, whose type rules enforce a subtype or supertype relationship between a qualified and unqualified type, suffice for some uses. Even these type systems can benefit from more sophisticated type rules, and more sophisticated and useful type systems require additional flexibility and expressiveness. Furthermore, an implementation of only part of a type system is impractical.

Appendix A

The Interning checker

This appendix lists for the source code of the Interning checker described in Section 5.4. The Interning checker's implementation makes use of many of the framework's features, including both declarative and procedural specification of type qualifiers, type introduction, and type rules described in Sections 4.3, 4.4, and 4.5, and the flow-sensitive type inference described in Section 4.8.

A.1 Qualifier declaration: `@Interned`

```
package checkers.quals;

import static java.lang.annotation.ElementType.*;

import java.lang.annotation.*;

import checkers.interned.InternedChecker;
import checkers.metaquals.*;
import checkers.types.AnnotatedTypeMirror.AnnotatedPrimitiveType;

import com.sun.source.tree.LiteralTree;

/**
 * Indicates that a variable has been interned, i.e., that the variable refers
 * to the canonical representation of an object.
 *
 * <p>
 *
 * This annotation is associated with the {@link InternedChecker}.
 *
 * @see InternedChecker
 * @manual #interned Interned Checker
 */
@Documented
@TypeQualifier
@Retention(RetentionPolicy.RUNTIME)
@Target({FIELD, LOCAL_VARIABLE, METHOD, PARAMETER, TYPE})
@ImplicitFor(
    treeClasses={LiteralTree.class},
    typeClasses={AnnotatedPrimitiveType.class})
public @interface Interned {

}
```

A.2 Compiler interface

```
package checkers.interned;

import checkers.basetype.*;
import checkers.metaquals.TypeQualifiers;
import checkers.quals.Interned;
import checkers.source.*;

import javax.annotation.processing.*;
import javax.lang.model.*;

/**
 * A typechecker plug-in for the {@link checkers.quals.Interned} qualifier that
 * finds (and verifies the absence of) equality-testing and interning errors.
 *
 * <p>
 *
 * The {@link checkers.quals.Interned} annotation indicates that a variable
 * refers to the canonical instance of an object, meaning that it is safe to
 * compare that object using the "==" operator. This plugin suggests using "=="
 * instead of ".equals" where possible, and warns whenever "==" is used in cases
 * where one or both operands are not {@link checkers.quals.Interned}.
 *
 * @manual #interned Interned checker
 */
@SupportedAnnotationTypes({"*"})
@SupportedSourceVersion(SourceVersion.RELEASE_7)
@SupportedLintOptions({"dotequals", "flow"})
@SuppressWarningsKey("interned")
@TypeQualifiers({ Interned.class })
public final class InternedChecker extends BaseTypeChecker { }
```

A.3 Visitor for type rules

```
package checkers.interned;

import java.util.*;

import checkers.source.*;
import checkers.basetype.*;
import checkers.types.*;
import checkers.util.*;
import com.sun.source.tree.*;
import com.sun.source.util.*;

import javax.lang.model.element.*;
import javax.lang.model.type.*;
import static javax.lang.model.util.ElementFilter.*;

/**
 * A type-checking visitor for the {@link checkers.quals.Interned} type
 * qualifier that uses the {@link BaseTypeVisitor} implementation. This visitor
 * reports errors or warnings for violations for the following cases:
 *
 * <ul>
 *  <li>if both sides of a "==" or "!=" comparison are not Interned (error
 *  "not.interned")</li>
 *  <li>if the receiver and argument for a call to an equals method are both
 *  Interned (optional warning "unnecessary.equals")</li>
 * </ul>
 *
 *
 * @see BaseTypeVisitor
 */
public class InternedVisitor extends BaseTypeVisitor<Void, Void> {
```

```
/** The interned annotation. */
private final AnnotationMirror INTERNED;
private final AnnotatedTypeMirror INTERNED_OBJECT;

/**
 * Creates a new visitor for type-checking {@link checkers.quals.Interned}.
 *
 * @param checker the checker to use
 * @param root the root of the input program's AST to check
 */
public InternedVisitor(InternedChecker checker, CompilationUnitTree root) {
    super(checker, root);
    this.INTERNED = annoFactory.fromName("checkers.quals.Interned");
    INTERNED_OBJECT = AnnotatedTypeMirror.createType(types.getDeclaredType(
                elements.getTypeElement("java.lang.Object")),
                checker.getProcessingEnvironment(), factory);
    INTERNED_OBJECT.addAnnotation(INTERNED);
}

@Override
public Void visitBinary(BinaryTree node, Void p) {

    // No checking unless the operator is "==" or "!=".
    if (!(node.getKind() == Tree.Kind.EQUAL_TO
            node.getKind() == Tree.Kind.NOT_EQUAL_TO))
        return super.visitBinary(node, p);

    Tree leftOp = node.getLeftOperand(), rightOp = node.getRightOperand();

    // Check passes if one arg is null.
    if (leftOp.getKind() == Tree.Kind.NULL_LITERAL
            rightOp.getKind() == Tree.Kind.NULL_LITERAL)
        return super.visitBinary(node, p);

    // Heuristically check that the comparison is the member of a class of
    // comparisons that should be skipped.
    if (suppressByHeuristic(node))
        return super.visitBinary(node, p);

    AnnotatedTypeMirror left = factory.getAnnotatedType(leftOp);
    AnnotatedTypeMirror right = factory.getAnnotatedType(rightOp);
    // Check passes due to auto-unboxing.
    if (left.getKind().isPrimitive()  right.getKind().isPrimitive())
        return super.visitBinary(node, p);

    if (!checker.isSubtype(INTERNED_OBJECT, left))
        checker.report(Result.failure("not.interned", left), leftOp);
    if (!checker.isSubtype(INTERNED_OBJECT, right))
        checker.report(Result.failure("not.interned", right), rightOp);

    return super.visitBinary(node, p);
}

@Override
public Void visitMethodInvocation(MethodInvocationTree node, Void p) {
    if (isInvocationOfEquals(node)) {
        AnnotatedTypeMirror recv = factory.getReceiver(node);
        AnnotatedTypeMirror comp = factory.getAnnotatedType(node.getArguments().get(0));

        if (this.checker.getLintOption("dotequals", true)
                && checker.isSubtype(INTERNED_OBJECT, recv)
                && checker.isSubtype(INTERNED_OBJECT, comp))
            checker.report(Result.warning("unnecessary.equals"), node);
    }

    return super.visitMethodInvocation(node, p);
}

/**
 * Tests whether a method invocation is an invocation of
 * {@link Object#equals}.
 *
 * @param node a method invocation node
 * @return true iff {@code node} is a invocation of {@code equals()}
 */
private boolean isInvocationOfEquals(MethodInvocationTree node) {
    ExecutableElement method = TreeUtils.elementFromUse(node);
    return (method.getParameters().size() == 1
            && method.getReturnType().getKind() == TypeKind.BOOLEAN
            && method.getSimpleName().contentEquals("equals"));
}

/**
 * Heuristically determines whether checking for a particular comparison
 * should be suppressed. Specifically, this method tests the following:
 *
 * <ul>
 * <li>the comparison is a == comparison, and</li>
 *
 * <li>it is the test of an if statement that's the first statement in the method,
 * and</li>
 *
 * <li>one of the following is true:
 *
 * <ul>
 * <li>the method overrides {@link Comparator#compare}, the "then" branch
 * of the if statement returns zero, and the comparison tests equality of
 * the method's two parameters</li>
 *
 * <li>the method overrides {@link Object#equals(Object)} and the
 * comparison tests "this" against the method's parameter</li>
 * </ul>
 *
 * </li>
 * </ul>
 *
 * @param node the comparison to check
 * @return true if one of the supported heuristics is matched, false
 *         otherwise
 */
private boolean suppressByHeuristic(final BinaryTree node) {

    // Only valid if called on an == comparison.
    if (node.getKind() != Tree.Kind.EQUAL_TO)
        return false;

    Tree left = node.getLeftOperand();
    Tree right = node.getRightOperand();

    // Only valid if we're comparing identifiers.
    if (!(left.getKind() == Tree.Kind.IDENTIFIER
            && right.getKind() == Tree.Kind.IDENTIFIER))
        return false;

    // If we're not directly in an if statement in a method (ignoring
    // parens and blocks), terminate.
    if (!Heuristics.matchParents(getCurrentPath(), Tree.Kind.IF, Tree.Kind.METHOD))
        return false;

    // Determine whether or not the "then" statement of the if has a single
    // "return 0" statement (for the Comparator.compare heuristic).
    final boolean returnsZero =
        Heuristics.applyAt(getCurrentPath(), Tree.Kind.IF, new Heuristics.Matcher() {

            @Override
            public Boolean visitIf(IfTree tree, Void p) {
                return visit(tree.getThenStatement(), p);
            }

            @Override
            public Boolean visitBlock(BlockTree tree, Void p) {
                if (tree.getStatements().size() > 0)
                    return visit(tree.getStatements().get(0), p);
                return false;
            }

            @Override
            public Boolean visitReturn(ReturnTree tree, Void p) {
                ExpressionTree expr = tree.getExpression();
                return (expr != null &&
                        expr.getKind() == Tree.Kind.INT_LITERAL &&
                        ((LiteralTree)expr).getValue().equals(0));
            }
        });


    ExecutableElement enclosing =
        TreeUtils.elementFromDeclaration(visitorState.getMethodTree());
    assert enclosing != null;

    Element lhs = TreeUtils.elementFromUse((IdentifierTree)left);
    Element rhs = TreeUtils.elementFromUse((IdentifierTree)right);

    if (returnsZero && overrides(enclosing, "java.util.Comparator", "compare")) {
        assert enclosing.getParameters().size() == 2;
        Element p1 = enclosing.getParameters().get(0);
        Element p2 = enclosing.getParameters().get(1);
        return (p1.equals(lhs) && p2.equals(rhs))
             (p2.equals(lhs) && p1.equals(rhs));
    } else if (overrides(enclosing, "java.lang.Object", "equals")) {
        assert enclosing.getParameters().size() == 1;
        Element param = enclosing.getParameters().get(0);
        Element thisElt = getThis(this.getCurrentPath());
        assert thisElt != null;
        return (thisElt.equals(lhs) && param.equals(rhs))
             (param.equals(lhs) && thisElt.equals(rhs));
    }

    return false;
}

/**
 * Determines the element corresponding to "this" inside a scope.
 *
 * @param path the path to a tree inside the desired scope
 * @return the element corresponding to "this" in the scope of the tree
 *         given by {@code path}
 */
private final Element getThis(TreePath path) {
    for (Element e : trees.getScope(path).getLocalElements())
        if (e.getSimpleName().contentEquals("this"))
            return e;
    return null;
}

/**
 * Determines whether or not the given element overrides the named method in
 * the named class.
 *
 * @param e an element for a method
 * @param clazz the name of a class
 * @param method the name of a method
 * @return true if the method given by {@code e} overrides the named method
 *         in the named class; false otherwise
 */
private final boolean overrides(
        ExecutableElement e, String clazz, String method) {

    // Get the element named by "clazz".
    TypeElement comp = elements.getTypeElement(clazz);
    if (comp == null) return false;
```

```
        // Check all of the methods in the class for name matches and overriding.
        for (ExecutableElement elt : methodsIn(comp.getEnclosedElements()))
            if (elt.getSimpleName().contentEquals(method)
                    && elements.overrides(e, elt, comp))
                return true;

        return false;
    }
}
```

A.4 Qualifier introduction

```
package checkers.interned;

import static java.util.Collections.singleton;

import javax.lang.model.element.*;

import checkers.basetype.BaseTypeChecker;
import checkers.flow.Flow;
import checkers.quals.Interned;
import checkers.types.*;
import static checkers.types.AnnotatedTypeMirror.*;

import com.sun.source.tree.*;

/**
 * An {@link AnnotatedTypeFactory} that accounts for the properties of the
 * Interned type system. This type factory will add the {@link Interned}
 * annotation to a type if the input:
 *
 * <ul>
 * <li>is a String literal
 * <li>is a class literal
 * <li>is an enum constant
 * <li>has a primitive type
 * <li>has the type java.lang.Class
 * <li>is a call to the method {@link String#intern()}
 * </ul>
 */
public class InternedAnnotatedTypeFactory extends AnnotatedTypeFactory {
```

```
/** Adds annotations from tree context before type resolution. */
private final TreeAnnotator treeAnnotator;

/** Adds annotations from the resulting type after type resolution. */
private final TypeAnnotator typeAnnotator;

/** The {@link Interned} annotation. */
final AnnotationMirror INTERNED;

/** Flow-sensitive qualifier inference. */
private final Flow flow;

/** An element of the class {@code java.lang.Class} */
private final TypeElement elementOfClass;

/** An element of class {@code java.lang.String} */
private final TypeElement elementOfString;
/**
 * Creates a new {@link InternedAnnotatedTypeFactory} that operates on a
 * particular AST.
 *
 * @param checker the checker to use
 * @param root the AST on which this type factory operates
 */
public InternedAnnotatedTypeFactory(InternedChecker checker,
    CompilationUnitTree root) {
    super(checker, root);
    this.INTERNED = annotations.fromName("checkers.quals.Interned");
    this.elementOfClass = elements.getTypeElement("java.lang.Class");
    this.elementOfString = elements.getTypeElement("java.lang.String");

    this.treeAnnotator = new TreeAnnotator(checker);
    this.typeAnnotator = new InternedTypeAnnotator(checker);

    this.flow = new Flow(checker, root, singleton(INTERNED), this);
    if (checker.getLintOption("flow", true)) flow.scan(root, null);
}

@Override
protected void annotateImplicit(Element elt, AnnotatedTypeMirror type) {
    typeAnnotator.visit(type);
}
```

```
@Override
protected void annotateImplicit(Tree tree, AnnotatedTypeMirror type) {
    treeAnnotator.visit(tree, type);
    final AnnotationMirror result = flow.test(tree);
    if (result != null)
        type.addAnnotation(result);
    typeAnnotator.visit(type);
}

/**
 * A class for adding annotations to a type after initial type resolution.
 */
private class InternedTypeAnnotator extends TypeAnnotator {

    /** Creates an {@link InternedTypeAnnotator} for the given checker. */
    InternedTypeAnnotator(BaseTypeChecker checker) {
        super(checker);
    }

    @Override
    public Void visitDeclared(AnnotatedDeclaredType t, Void p) {

        // Enum types and constants: add an @Interned annotation.
        Element elt = t.getUnderlyingType().asElement();
        assert elt != null;
        if ((elt.getKind() == ElementKind.ENUM)
                 (elementOfClass.equals(elt)))
            t.addAnnotation(INTERNED);

        return super.visitDeclared(t, p);
    }

    @Override
    public Void visitExecutable(AnnotatedExecutableType t, Void p) {

        // Annotate the java.lang.String.intern() method.
        ExecutableElement method = t.getElement();
        if (method.getSimpleName().contentEquals("intern")
                && elementOfString.equals(method.getEnclosingElement()))
            t.getReturnType().addAnnotation(INTERNED);

        return super.visitExecutable(t, p);
    }
    }

    @Override
    public AnnotatedPrimitiveType getUnboxedType(AnnotatedDeclaredType type) {
        AnnotatedPrimitiveType primitive = super.getUnboxedType(type);
        primitive.addAnnotation(INTERNED);
        return primitive;
    }
}
```

Bibliography

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

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

[3] Chris Andreae, James Noble, Shane Markstrum, and Todd Millstein. A framework for implementing pluggable type systems. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2006)*, pages 57–74, Portland, OR, USA, October 24–26, 2006.

[4] Shay Artzi, Jaime Quinonez, Adam Kieżun, and Michael D. Ernst. A formal definition and evaluation of parameter immutability., December 2007. Under review.

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

[6] Adrian Birka and Michael D. Ernst. A practical type system and language for reference immutability. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2004)*, pages 35–49, Vancouver, BC, Canada, October 26–28, 2004.

[7] Joshua Bloch. JSR 175: A metadata facility for the Java programming language. `http://jcp.org/en/jsr/detail?id=175`, September 30, 2004.

[8] Daniel Bonniot, Bryn Keller, and Francis Barber. *The Nice user's manual*, 2003. `http://nice.sourceforge.net/`.

[9] Gilad Bracha. Pluggable type systems. In *OOPSLA Workshop on Revival of Dynamic Languages*, Vancouver, BC, Canada, October 2004.

[10] Brendon Cahoon and Kathryn S. McKinley. Data flow analysis for software prefetching linked data structures in Java. In *10th International Conference on Parallel Architectures and Compilation Techniques*, pages 280–291, Barcelona, Spain, September 10–12, 2001.

[11] Patrice Chalin and Perry R. James. Non-null references by default in Java: Alleviating the nullity annotation burden. In *ECOOP 2007 — Object-Oriented Programming, 21st European Conference*, pages 227–247, Berlin, Germany, August 1–3, 2007.

[12] Brian Chin, Shane Markstrum, and Todd Millstein. Semantic type qualifiers. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation*, pages 85–95, Chicago, IL, USA, June 13–15, 2005.

[13] Telmo Luis Correa Jr., Jaime Quinonez, and Michael D. Ernst. Tools for enforcing and inferring reference immutability in Java. In *Companion to Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2007)*, pages 866–867, Montréal, Canada, October 23–25, 2007.

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

[15] Robert DeLine and Manuel Fähndrich. Enforcing high-level protocols in low-level software. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, pages 59–69, Snowbird, UT, USA, June 20–22, 2001.

[16] 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.

[17] Eiffel: Analysis, design and programming language. Standard ECMA-367, June 2006. 2nd edition.

[18] Torbjörn Ekman and Görel Hedin. The JastAdd extensible Java compiler. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2007)*, pages 1–18, Montréal, Canada, October 23–25, 2007.

[19] Torbjörn Ekman and Görel Hedin. Pluggable checking and inferencing of non-null types for Java. *Journal of Object Technology*, 6(9):455–475, October 2007.

[20] Martin Elsman, Jeffrey S. Foster, and Alexander Aiken. *Carillon — A System to Find Y2K Problems in C Programs*, July 30, 1999.

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

[22] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):99–123, February 2001. A previous version appeared in *ICSE '99, Proceedings of the 21st International Conference on Software Engineering*, pages 213–224, Los Angeles, CA, USA, May 19–21, 1999.

[23] Michael D. Ernst and Danny Coward. JSR 308: Annotations on Java types. `http://pag.csail.mit.edu/jsr308/`, November 9, 2007.

[24] A. P. Ershov. On programming of arithmetic operations. *Communications of the ACM*, 1(8):3–6, August 1958.

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

[26] David Evans and David Larochelle. Improving security using extensible lightweight static analysis. *IEEE Software*, 19(1):42–51, 2002.

[27] Manuel Fähndrich and K. Rustan M. Leino. Declaring and checking non-null types in an object-oriented language. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2003)*, pages 302–312, Anaheim, CA, USA, November 6–8, 2003.

[28] Jean-Christophe Filliâtre and Sylvain Conchon. Type-safe modular hash-consing. In *ML '06: Proceedings of the 2006 workshop on ML*, pages 12–19, Portland, OR, USA, September 16, 2006.

[29] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, pages 234–245, Berlin, Germany, June 17–19, 2002.

[30] Philip W. L. Fong. Pluggable verification modules: An extensible protection mechanism for the JVM. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2004)*, pages 404–418, Vancouver, BC, Canada, October 26–28, 2004.

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

[32] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification.* Addison Wesley, Boston, MA, third edition, 2005.

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

[34] David Greenfieldboyce and Jeffrey S. Foster. Type qualifier inference for Java. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2007)*, pages 321–336, Montréal, Canada, October 23–25, 2007.

[35] David Hovemeyer, Jaime Spacco, and William Pugh. Evaluating and tuning a static analysis to find null pointer bugs. In *ACM SIGPLAN/SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE 2005)*, pages 13–19, Lisbon, Portugal, September 5–6, 2005.

[36] Rob Johnson and David Wagner. Finding user/kernel pointer bugs with type inference. In *13th USENIX Security Symposium*, pages 119–134, San Diego, CA, USA, August 11–13, 2004.

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

[38] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: A behavioral interface specification language for Java. *ACM SIGSOFT Software Engineering Notes*, 31(3), March 2006.

[39] 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`, August 20, 2007.

[40] Darko Marinov and Robert O'Callahan. Object equality profiling. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2003)*, pages 313–325, Anaheim, CA, USA, November 6–8, 2003.

[41] Shane Markstrum, Daniel Marino, Matthew Esquivel, and Todd Millstein. Practical enforcement and testing of pluggable type systems. Technical Report CSD-TR-080013, UCLA, April 2008.

[42] Jens Palsberg and Peter Ørbæk. Trust in the λ-calculus. In *Proceedings of the Second International Symposium on Static Analysis, SAS '95*, pages 314–329, Glasgow, UK, September 25–27, 1995.

[43] Matthew M. Papi, Mahmood Ali, Telmo Luis Correa Jr., Jeff H. Perkins, and Michael D. Ernst. Pluggable type-checking for custom type qualifiers in Java. Technical Report

MIT-CSAIL-TR-2007-047, MIT Computer Science and Artificial Intelligence Laboratory, Cambridge, MA, September 17, 2007.

[44] Matthew M. Papi, Mahmood Ali, Telmo Luis Correa Jr., Jeff H. Perkins, and Michael D. Ernst. Practical pluggable types for Java. In *ISSTA 2008, Proceedings of the 2008 International Symposium on Software Testing and Analysis*, Seattle, WA, USA, July 22–24, 2008.

[45] Jeff H. Perkins and Michael D. Ernst. Efficient incremental algorithms for dynamic detection of likely invariants. In *Proceedings of the ACM SIGSOFT 12th Symposium on the Foundations of Software Engineering (FSE 2004)*, pages 23–32, Newport Beach, CA, USA, November 2–4, 2004.

[46] 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.

[47] Sara Porat, Marina Biberstein, Larry Koved, and Bilba Mendelson. Automatic detection of immutable fields in Java. In *CASCON*, Mississauga, Ontario, Canada, November 13–16, 2000.

[48] Jaime Quinonez, Matthew S. Tschantz, and Michael D. Ernst. Inference of reference immutability. In *ECOOP 2008 — Object-Oriented Programming, 22nd European Conference*, Paphos, Cyprus, July 9–11, 2008.

[49] Umesh Shankar, Kunal Talwar, Jeffrey S. Foster, and David Wagner. Detecting format string vulnerabilities with type qualifiers. In *10th USENIX Security Symposium*, Washington, DC, USA, August 15–17, 2001.

[50] Mats Skoglund and Tobias Wrigstad. A mode system for read-only references in Java. In *FTfJP'2001: 3rd Workshop on Formal Techniques for Java-like Programs*, Glasgow, Scotland, June 18, 2001.

[51] Matthew S. Tschantz. Javari: Adding reference immutability to Java. Technical Report MIT-CSAIL-TR-2006-059, MIT Computer Science and Artificial Intelligence Laboratory, Cambridge, MA, September 5, 2006.

[52] Matthew S. Tschantz and Michael D. Ernst. Javari: Adding reference immutability to Java. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2005)*, pages 211–230, San Diego, CA, USA, October 18–20, 2005.

[53] Mandana Vaziri, Frank Tip, Stephen Fink, and Julian Dolby. Declarative object identity using relation types. In *ECOOP 2007 — Object-Oriented Programming, 21st European Conference*, Berlin, Germany, August 1–3, 2007.

[54] Dennis M. Volpano and Geoffrey Smith. A type-based approach to program security. In *TAPSOFT '97: Theory and Practice of Software Development, 7th International Joint Conference CAAP/FASE*, pages 607–621, Lille, France, April 14–18, 1997.

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

[56] Olivier Zendra and Dominique Colnet. Towards safer aliasing with the Eiffel language. In *Interontinental Workshop on Aliasing in Object-Oriented Systems*, pages 153–154, Lisbon, Portugal, June 15, 1999.

[57] Yoav Zibin, Alex Potanin, Mahmood Ali, Shay Artzi, Adam Kieżun, and Michael D. Ernst. Object and reference immutability using Java generics. In *ESEC/FSE 2007: Proceedings of the 11th European Software Engineering Conference and the 15th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, Dubrovnik, Croatia, September 5–7, 2007.