Practical Pluggable Types for Java

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

Abstract

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 5 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.

Categories and subject descriptors: D3.3 [Programming Languages]: Language Constructs and Features—*data types and structures*; F3.1 [Logics and Meanings of Programs]: Specifying and Verifying and Reasoning about Programs; D1.5 [Programming Techniques]: Object-oriented Programming
General terms: Languages, Theory
Keywords: annotation, bug finding, case study, compiler, flow-sensitivity, IGJ, immutable, intern, Java, javac, Javari, NonNull, pluggable type, polymorphism, readonly, type qualifier, type system, verification

1. Introduction

A static type system helps programmers to detect and prevent errors. However, a language's built-in type system does not help to detect and prevent *enough* errors, because it cannot express certain important invariants. A user-defined, or pluggable, type system enriches the built-in type system by expressing extra information about types via type qualifiers. Example qualifiers include `nonnull`, `readonly`, `interned`, and `tainted`. Pluggable types permit more expressive compile-time checking and guarantee the absence of additional errors.

A pluggable type framework serves two key constituencies. It enables a *programmer* to write type qualifiers in a program and to run a type checker that verifies that the program respects the type system. It enables a *type system designer* to define type qualifiers, to specify their semantics, and to create the checker used by the programmer.

Programmers wish to improve the quality of their code without disrupting their workflow. To be useful to programmers, a pluggable type framework should be integrated with the programming language, run-time system, and toolchain. The qualifiers should be a part of the programming language that is supported by every tool and is preserved in compiled executables to the same extent that ordinary types are. An unmodified compiler should be capable of checking the pluggable type system.

Type system designers wish to evaluate and deploy their type systems: a type system is valuable only if it helps programmers, so using a type system in practice is an essential way to gain insight. To be useful to type system designers, a pluggable type framework should make it easy to define simple type systems and possible to define powerful type systems. The framework should declaratively support common type system idioms such as the type hierarchy, implicit type qualifiers, flow-sensitivity, and parametric polymorphism over both types and qualifiers. However, the framework should also support procedural specification of features specific to the type system. Furthermore, the declarative and procedural mechanisms should be tightly coupled in a single language. When expressiveness and conciseness are in conflict, expressiveness is more important. A well-known type system is easy enough to understand and implement, whatever the framework, but novel type systems will inevitably require new functionality, and the framework should not be tuned to the type systems of yesterday.

Despite considerable interest in user-defined type qualifiers, previous frameworks have been too inexpressive, unscalable, or incompatible with existing languages or tools. This has hindered the evaluation, understanding, and uptake of pluggable types.

We have implemented a pluggable type framework, called the Checker Framework (because it is used for writing type-checkers), that satisfies these goals in the context of a mainstream object-oriented language, Java. Experience with the framework indicates that it is successful in satisfying both programmers and type system designers. For both constituencies, the tools were natural to use. For programmers, checkers built using the framework have been run on over 600 KLOC, revealing many errors and verifying the absence of further such problems. For type system designers, the case studies revealed new insights, even about known type systems.

The Checker Framework provides a backward-compatible syntax for expressing type qualifiers, in the form of an extension to

ISSTA'08, July 20–24, 2008, Seattle, Washington, USA.

the Java annotation system that is planned for inclusion in the Java 7 language. A programmer can write type qualifiers throughout a program as machine-checked documentation. The Checker Framework allows a type system designer to define new type qualifiers and their semantics in a declarative and/or procedural manner. The framework then creates a type checker in the form of a compiler plug-in. Our framework is compatible with Java, so it handles pluggable type systems that refine, not incompatibly change, Java's built-in type system.

To demonstrate the practicality of the Checker Framework, we have written type-checkers for five different type systems. The Basic checker enforces only type hierarchy 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.

We have performed case studies with each checker. These case studies evaluate the Java source code syntax for qualifiers, the ease of creating a type-checker, and the quality and usability of the resulting checker. Furthermore, the case studies evaluate the type systems themselves. The Interning type system is novel. The others were previously known, but the realism of our evaluation has led to new observations and improvements.

All of the tools described in this paper are publicly available and documented at `http://pag.csail.mit.edu/jsr308/`. These tools can aid practitioners and can permit researchers to experiment with new type systems in the context of a realistic language.

This paper first describes the Checker Framework from the perspective of a programmer (Section 2) and of a type system designer (Section 3). Then, Sections 4–9 summarize our case studies. Section 10 discusses related work. Finally, the paper concludes by presenting lessons learned (Section 11) and recapping our research contributions (Section 12).

2. The programmer's view of a checker

This section describes the Checker Framework from a programmer's point of view. Section 3 describes it from a type system designer's point of view. This research paper is not a manual; for detailed usage information, see the Checker Framework Manual, available at `http://pag.csail.mit.edu/jsr308/`.

The Checker Framework seamlessly integrates type-checking with the compilation cycle. Programmers write type qualifiers using a backwards-compatible extension to Java's annotation syntax. A checker runs as a compiler plug-in for the javac Java compiler.

2.1 Using a checker to detect software errors

Our framework uses Java's standard compiler flag, `-processor`, for invoking an annotation processor [6]:

```
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 [25, 6]. As a result, checker errors/warnings are formatted like compiler error/warnings, and the compiler is aware of them when determining whether to continue compilation (e.g., 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.

2.2 Type qualifier syntax

Java's annotation system [4] permits programmers to attach metadata called annotations to declarations. This is inadequate for expressing type qualifiers on generics, casts, and a host of other locations. We have defined an extension that permits annotations to appear on nearly any use of a type [10]. In brief, the changes to the Java language grammar are:

1. A type annotation may be written before any type, as in `@NonNull String`.
2. An array type is annotated on the brackets `[]` that indicate the array, separately from an annotation on the element type.
3. A method receiver (`this`) type is annotated just after the parameter list.

Examples of the new syntax are:

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

The syntax extension specifies where a programmer may write annotations, but not their semantics (that is the role of the checker).

We also defined a backward-compatible class file format for storing type annotations. For example, a qualifier on a field's declared type is stored in an attribute of the field. The qualifiers affect neither the bytecode instructions produced by the Java compiler nor the Java Virtual Machine. Any conformant compiler, even if it is not using a checker plug-in, will preserve the annotations to the compiled class file for use by subsequent tools — for instance, type-checking against a compiled library or bytecode verification. We have built the publicly-available Annotation File Utilities for inserting annotations in, and extracting annotations from, source code and class files.

To aid in migration, our implementation permits annotations to be enclosed in comments, as in `List</*@NonNull*/ String>`. Source code that uses this syntax can be processed by compilers for earlier versions of Java.

Our syntax proposal has been assigned the Sun codename "JSR 308" [10] 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 Section 3, that requires no compiler changes beyond these 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 tools; by contrast, our implementation works with the NetBeans IDE and the 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. Checker Framework

The Checker Framework enables a type system designer to define the rules for a type system. The Checker Framework then creates a type-checking compiler plug-in (for short, a checker) that applies the rules to a program being compiled. This section describes features of the Checker Framework that support this process.

3.1 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.

(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 semantics 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 `@NonNull` types 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, etc.

Sections 3.2–3.5 describe how the Checker Framework supports defining these four components of a type system. The Checker Framework supports parametric polymorphism over both (qualified) types and type qualifiers (Section 3.6), and it performs flow-sensitive inference of type qualifiers (Section 3.7).

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 to extend the framework's default functionality. 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 [1] 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.

The Checker Framework also provides a representation of annotated types, `AnnotatedTypeMirror`, that extends the standard `TypeMirror` interface 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.

3.2 Type qualifiers and hierarchy

Type qualifiers are defined as Java annotations [6], extended as described in Section 2.2.

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.

`@SubtypeOf` accepts multiple annotation classes as an argument, permitting the type hierarchy to be an arbitrary DAG. For example, Figure 6 shows that in the IGJ type system (Section 9), `@Mutable` and `@Immutable` induce two mutually exclusive subtypes of the `@ReadOnly` qualifier.

The `@DefaultQualifier` meta-annotation indicates which qualifier implicitly appears on unannotated types. This may ease the annotation burden or provide backward compatibility with unannotated programs.

A type system whose default is not the root of the qualifier hierarchy (such as 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. The rule ensures backward compatibility while maximizing the number of possible type parameters that a client may use.

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 qualifier 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 dates is a subtype of a readonly list of readonly dates.

3.3 Implicit annotations: qualifier introduction

Certain constructs should be treated as having a type qualifier even when the programmer has not written one. For example, in the Nullness type system, string literals implicitly have the type `@NonNull String`. In the Interning type system, string literals 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), of symbols (such as exception parameters), and/or of expressions (such as string literals) that should be annotated.

Type system designers can augment the declarative syntax by additionally overriding the `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`.

Implicit annotations are distinct from, and take precedence over, the `@DefaultQualifier` annotation of Section 3.2.

Implicit annotations could be handled as a special case of type rules (Section 3.4), but we found it more natural to separate them, as is also often done in formal expositions of type systems.

3.4 Defining type rules

A type system's rules define which operations are forbidden. The Checker Framework builds in checks of the type hierarchy. Here are examples. It checks that, in every assignment and pseudo-assignment, the lhs is a supertype of (or the same type as) the rhs; this assignment is forbidden (assuming the obvious variable declarations):

```
myNonNullObject = myObject;  // invalid assignment
```

It checks the validity of overriding and subclassing. It prohibits inconsistent annotations at a single location.

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 that traverse the program's AST.

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.

3.5 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 declaratively associate type qualifiers with a checker via the `@TypeQualifiers` annotation. Other annotations configure the plug-in's command-line options and the anno-

tations that suppress its warnings. For details, see the Checker Framework manual.

3.6 Parametric polymorphism

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

3.6.1 Type polymorphism

As noted in Section 2.2, a programmer can annotate generic type arguments in a natural way, which is critical for real Java programs.

The subtyping rules of Section 3.4 fully support qualified generic types.

The Checker Framework performs generic type inference [14, §15.12.2.7] for invocation of generic methods. It infers the most restrictive qualified type arguments based on actual arguments. If the type variable is not used in a method parameter, as in calls to `Collections.emptyList`, then inference uses the assignment context. If that too fails to resolve the type argument, the default is `Object`. Generic type inference eliminated dozens of false positive warnings in our case studies.

The Checker Framework also performs capture conversion [14, §15.25], for example when determining the type of a conditional expression (`?:`) from the types of its true and false expressions. The framework warns when, due to Java's invariant generic subtyping, the expression has no qualified generic type.

3.6.2 Qualifier polymorphism

The Checker Framework supports type qualifier polymorphism for methods, limited to a single qualifier variable. (We have found one variable 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:

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

Then, a programmer can use the marked annotation as a type qualifier variable. For example, `Class.cast` returns null iff its 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.

3.7 Flow-sensitive type qualifier inference

The Checker Framework performs flow-sensitive intraprocedural qualifier inference. The inference may compute a more specific type (that is, a subtype) for a reference than that given in its declaration. For example, the Nullness checker (Section 6) 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. The inference often eliminates the need to annotate method bodies, since variables that are declared without annotations are treated as (for example) non-null where appropriate.

Checker & Program	Size Files	Lines	ALocs	Ann.s	Err-ors	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

Figure 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 the checker implementation.

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 way. For the GEN portion, a reference is known to be non-null (e.g.) after a null check in an assert statement or a conditional, or after a non-null value is assigned to it. For the KILL portion, a reference is no longer non-null (e.g.) 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 abstractions pertaining to dataflow (e.g., the `split` and `merge` methods handle GEN-KILL sets at branches). In addition, a type system designer can specialize the analyses by extending visitor methods or the higher-level abstractions. 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.

4. Experiments

This section summarizes case studies that evaluate our designs and implementations. Most of the case studies (~400KLOC) were completed in summer 2007, and technical reports give additional details [21, 20] such as many examples of specific errors found.[1] Space limits prevent discussing subsequent case studies. As one example, the author of FreePastry (`http://freepastry.rice.edu/`, 1084 files, 209 KLOC) used the Interning checker to find problems in his code.

Figure 1 lists the subject programs. The annotation file utili-

[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.

ties (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], a dynamic invariant detector. JOlden is a benchmark suite[4]. The partial JDK is several packages from Sun's implementation[5]. TinySQL is a library implementing the SQL query language[6]. Htmlparser is a library for parsing HTML documents[7]. SVNKit is a client for the Subversion revision control system[8]. Lucene is a text search engine library[9].

The sizes in Figure 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.

4.1 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, rather than by bug-finding that aims to discover a few "low-hanging fruit" errors, albeit with less programmer effort. (See Section 6 for an empirical comparison of the verification and bug-finding approaches.) Therefore, the number of errors reported in Figure 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. Mere "bad code smells" count as false positives, even though refactoring would improve the code.

Warnings that cannot be eliminated via annotation, but that cannot cause incorrect user-visible behavior, count as false positives. The programmer suppressed each false positive with a `@SuppressWarnings` annotation or, for the Nullness checker, an assertion `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 pos
}
```

`Map.get` is specified to possibly return null (it does so if the key is not found in the map); however, in the above code the `Map.get` return value is non-null, because `key` is in the map. The other 6 false positives were weaknesses in a checker implementation.

Our studies analyze existing programs. By contrast, writing new programs matched to the checker's capabilities is likely to be much easier and to require fewer annotations. We note possible bias in that a few of the subject programs are the checkers themselves, though the programmers wrote annotations only after the checker was operational.

[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/
[7] http://htmlparser.sourceforge.net/
[8] http://svnkit.com/
[9] http://lucene.apache.org/java/docs/index.html

Checker	Total	Type intro.	Flow	Type rules	Compiler i/f
Basic	36	0	0	0	36
Nullness	472	165	250	49	8
Interning	186	53	0	125	8
Javari	418	278	n/a	99	41
IGJ	409	328	n/a	0	81

Figure 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.

4.2 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. Figure 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 [11] to the Nullness checker took about 1 hour. It took about 2 hours to generalize the Nullness-specific flow-sensitive type qualifier inference [21] into the framework feature of Section 3.7.

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. The Basic checker for simple type systems

The Basic checker performs only checks related to the type hierarchy (see Section 3.4). This is adequate for simple type systems and for prototyping.

The type system designer writes no code besides annotation definitions (which have empty bodies). The programmer names the checked annotations 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, type and qualifier polymorphism, and flow-sensitive inference.

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 3.1) in several

[10] However, we have since developed inference tools for the Javari and Nullness type systems. These tools, discussed in the Checker Framework manual, would further reduce the annotation burden, particularly for libraries and legacy code.

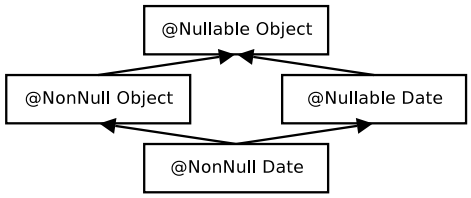

Figure 3: Type hierarchy for the Nullness type system. Java's `Object` is expressed as `@Nullable Object`. Programmers can omit most type qualifiers, thanks to careful choice of defaults.

phases, starting from an unannotated type provided by the underlying compiler. The `@Fully` type qualifier indicates that construction is complete. A `@Fully` annotated type is a subtype of a `@Partly` annotated type.

Then, a programmer annotated each use of `AnnotatedTypeMirror` 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 3.6.2). For instance, the component type of an array type has the same annotatedness as the array type.

6. Nullness checker for null pointer errors

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 3). 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 pointer exception.

The Nullness checker supports the `@Raw` type qualifier for partially-initialized objects [11]. (The `@Raw` type qualifier is unrelated to the raw types of Java generics.)

The Nullness checker's visitor class ("type rules" column of Figure 2) implements three type rules, for: dereferences of possibly-null expressions, implicit iteration over possibly-null collections in an enhanced `for` loop, and accessing a possibly-null array. The type introduction rules add the `@NonNull` annotation to literals (except `null` gets `@Nullable`), `new` expressions, and classes used for static member access (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 Figure 1, but these redundant tests and dead code are often correlated with other errors [16].

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. In the Nullness checker program, 39 of the 45 false positives (Figure 1) were due to complex nullness invariants, especially in AST node operations. Expressing such application invariants would require a substantially more sophisticated system, such as dependent types. The flexibility of the Checker Framework permits more sophisticated checks to be coded even if they are not expressible in the type system being checked. For example, the best type for `toArray` is both reflective and polymorphic, and checkers can treat it as such.

We evaluated our checker against the null pointer checks of several other static analysis tools, using the Lookup subject program.

	Nullable			NonNull			NNEL		
Program	Tot	Sig	Body	Tot	Sig	Body	Tot	Sig	Body
Annotation file utils	760	483	277	169	90	79	107	90	17
Lookup	382	301	81	80	33	47	35	33	2
Nullness checker	—			282	126	156	146	126	20

Figure 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").

Here are the results:

	errors		false	annotations
Tool	found	missed	warnings	written
Checker Framework	8	0	4	35
FindBugs	0	8	1	0
JLint	0	8	8	0
PMD	0	8	0	0

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.

6.1 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 in conjunction with the flow-sensitive type inference of Section 3.7. 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.

Non-null-by-default reduces clutter because non-null types are more prevalent, and it is good to bias programmers away from using nullable variables. Splint, Nice, JML, and Eiffel have adopted non-null-by-default semantics.

To evaluate the defaults, the programmer annotated subject programs multiple 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 [5] determined via manual inspection that about 3/4 of variables in JML code and the Eclipse compiler are dereferenced without being checked.

Figure 4 shows our results. The NNEL code was not just terser, but — more importantly — clearer to the programmers in our study. Reduced clutter directly contributes to readability. 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. A potential downside of non-uniform defaults is that an unannotated type such as "`Integer`" means different things in different parts of the code. Programmers did not find this to be a problem in practice, perhaps because programmers think of public declarations differently than private implementa-

tions. 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.

7. Interning checker for equality-testing errors

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

Interning yields both space and time benefits. However, misuse of interning can lead to bugs: use of == on distinct objects representing the same abstract value may return false, as in `new Integer(22) == new Integer(22)` which yields `false`.

The Interning type hierarchy is similar to that for NonNull (Figure 3). We believe that ours is the first formulation of a completely backward-compatible system for interning.

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
```

(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 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 Figure 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 only annotation that would otherwise be necessary in the JDK).

We evaluated the Interning checker by applying it to Daikon. Daikon is a good subject program because memory usage is the limiting factor in its scalability and because the programmers have spent considerable time and effort validating its use of interning, including 200 run-time checks and a lexical code analysis tool. Nonetheless, annotating only part of Daikon revealed 9 errors and 2 optimization opportunities.

The false positives in Figure 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 ==.

8. Javari checker for mutation errors

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

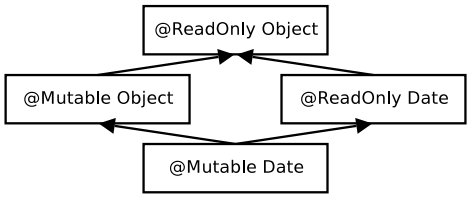

Figure 5: Type hierarchy for Javari's ReadOnly and Mutable type qualifiers.

Javari [3, 23] is an extension of the Java language that prevents mutation errors via specification and compile-time verification of immutability constraints. Figure 5 shows the type hierarchy. A reference of `@ReadOnly` type may not be used to mutate any part of its referent. The full type system is described elsewhere [23].

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 to a reference with ReadOnly type. 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", previously known as "RoMaybe"). The reasons these types are useful and necessary are explained elsewhere [23].

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 is reflective and modifies its argument exactly if the argument has greater size than the receiver.

The annotations did not clutter the code, because they appeared mostly on method signatures; leaving local variables unannotated (mutable) was usually sufficient. The few local variable annotations appeared at existing Java casts, where the type qualifier had to be made explicit; flow-sensitive analysis (Section 3.7) would have eliminated these.

The programmer was able to annotate more local variables in the Javari checker than in the JOlden benchmark, due to better encapsulation and more use of getter methods. Most of the annotations were `@ReadOnly` (288 annotations on classes, 514 annotations on libraries and interfaces). 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.

9. IGJ checker for mutation errors

Immutability Generic Java (IGJ) [27] is a Java language extension that expresses immutability constraints. Like the Javari language described in Section 8, 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, 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 checker has no special type rules, so it does not extend the visitor class. It defines the type hierarchy in the compiler

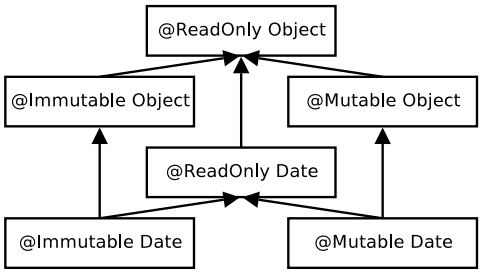

Figure 6: Partial type hierarchy for IGJ.

interface. The framework's subtype tests 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.

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.

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 a class 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 `List`s and `Array`s 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. Immutability types are a poor fit to such classes, but leaving them unannotated (the default is mutable, for backward compatibility with Java) worked well.

We gained some insights into how the type rules apply in practice. Less than 10% of classes had constructors calling setter methods. Programmers showed discipline regarding immutability 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 immutability parameter. The programmer used few mutable fields; one of the rare exceptions was a collection (in `SVNErrorCode`) that contains all `SVNErrorCode`s 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.

10. Related work

Space permits us to discuss only the most closely related work on pluggable type-checking/inference frameworks for Java and on the specific type systems used in our case studies.

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 indicate that this is an important and challenging goal.

The most direct comparison comes from the fact that JQual [15], JavaCOP [1], and the Checker Framework have all been used to implement the Javari [23] 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 properly 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 [2]. Another point of comparison is JavaCOP's Nullness type system. Recent work [19] 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, has fewer lines than ours (472 lines[11]), but lacks functionality present in ours such as support for generics and arrays, checking implicit dereferences in foreach loops and array accesses, customizable default annotation, 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, expressiveness, and semantics. We introduced 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 [1]; the declarative and procedural parts are not integrated as they are in the Checker Framework. 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 frame-

[11] Of the 472 lines, 127 consist only of "}", "`@Override`", or a `package` or `import` statement.

work 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 released[12] after the Checker Framework was, and after we published our case studies [21]. Markstrum et al. [19] report that the JavaCOP framework has recently acquired some of the features of the Checker Framework, such as flow-sensitive type inference[13] and integration with javac.[14] 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). 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. JavaCOP had a declarative syntax earlier than the Checker Framework, though the designs are rather different. 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 [19] suggest that may be the context in which JavaCOP really shines.

Inference.

JQual [15] 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. The Checker Framework does not infer annotations on signatures. 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. Our NNEL (NonNull Except Locals) approach can be viewed as being similar to, and about as effective as, local type inference — users can leave bodies largely unannotated. Our framework interfaces with scalable inference tools for the Nullness and Javari type systems.

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 [15, 2], so an experimental comparison is impossible.

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 [9] ran a significant number of null-checking tools and reports on their strengths and weaknesses; Chalin and James [5] give another recent survey. We mention four notable practical tools. ESC/Java [13] 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. This is more powerful than a type system, but suffers from scalability limitations. The JastAdd extensible Java compiler [7] includes a module for checking and inferencing of non-null types [8] (and JastAdd could theoretically be used as a framework to build other type systems). The JACK Java Annotation ChecKer [17] 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 [16] takes a very different approach. FindBugs uses an extremely coarse analysis that yields mostly false positives, then uses heuristics to discard reports about values that might result from infeasible paths.

Interning.

Vaziri et al. [24] 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 [18] 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 [26]. 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 [18, 12, 24] 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.

Javari.

Our implementation is the first checker for the complete Javari language [23, 22], and incorporates several improvements that are described in a technical report. There have been three previous attempts to implement Javari. Birka [3] 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 [1] and JQual [15] 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.

IGJ.

Our implementation is the second checker for the IGJ language. The previous IGJ dialect [27] 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>`.

11. 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 dialect of Javari [3], 106 KLOC in IGJ [27], 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.

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

[13] Publicly available and documented in our framework in February 2008.

[14] 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 processing.

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 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.

IGJ. Our subject programs used class, object, and reference immutability in different parts or for different classes, demonstrating the advantages of a rich immutability type system. 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 replacing elements but not insertion nor removal.

IGJ was inspired by Java's generics system. To our surprise, the programmer preferred the 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>>`. The programmer found prefix modifiers more natural, as in `@Mutable List<@ReadOnly Date>`.

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, such as 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, flow-sensitive inference eliminated only 3 annotations in the Interning case study.

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 arrays 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 [27], 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 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.

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 if generics would suffice. Qualifier polymorphism is part of 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. We have not yet enountered a need for multiple qualifier variables, much less for subtype constraints among them [15].

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 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.

Framework design. Integrating but decoupling the checker and the compiler yielded a workable, practical, and deployable system.

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.

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 generic types.

Inference. Inference of type annotations has the potential to greatly reduce the programmer 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., in Javari and IGJ).

Existing inference tools tend to scale poorly. After iterating for many months offering bug reports on every static non-null inference tool available, we wrote our own sound, dynamic nullable inference tool in a weekend. Just as the Checker Framework fills a need for type checking, there is a need for robust, scalable, expressive type frameworks that specifically support static inference.

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.

12. 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 program-

mers, 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 syntax is planned for inclusion in the Java 7 language under its Sun codename "JSR 308".

• 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 it 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 enforces only type hierarchy 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 relatively low annotation burden and manageable false positive rates.

• New insights about previously-known type systems (see Section 11).

• 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, and in the first author's thesis [20]. 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.

13. References

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

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

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

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

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

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

[7] Torbjörn Ekman and Görel Hedin. The JastAdd extensible Java compiler. In *OOPSLA*, pages 1–18, Oct. 2007.

[8] Torbjörn Ekman and Görel Hedin. Pluggable checking and inferencing of non-null types for Java. *J. Object Tech.*, 6(9):455–475, Oct. 2007.

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

[10] Michael D. Ernst. Annotations on Java types: JSR 308 working document. `http://pag.csail.mit.edu/jsr308/`, Nov. 12, 2007.

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

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

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

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

[15] David Greenfieldboyce and Jeffrey S. Foster. Type qualifier inference for Java. In *OOPSLA*, pages 321–336, Oct. 2007.

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

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

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

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

[20] Matthew Papi. Practical pluggable types for Java. Master's thesis, MIT Dept. of EECS, May 2008.

[21] 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 CSAIL, Sep. 17, 2007.

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

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

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

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

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

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