Compile-time Type-checking for Custom Type Qualifiers in Java

Matthew M. Papi Michael D. Ernst
MIT Computer Science and Artificial Intelligence Lab, Cambridge, MA, USA
{mpapi,mernst}@csail.mit.edu

Abstract

We have created a system that enables programmers to add custom type qualifiers to the Java language in a backward-compatible way. The system allows programmers to write type qualifiers in their programs and to create compiler plug-ins that enforce the semantics of these qualifiers at compile time. The system builds on existing Java tools and APIs, and on JSR 308.

As an example, we introduce a plug-in to Sun's Java compiler that uses our system to type-check the NonNull qualifier. Programmers can use the `@NonNull` annotation to prohibit an object reference from being null; then, by invoking a Java compiler with the NonNull plug-in, they can check for NonNull errors at compile time and rid their programs of null-pointer exceptions.

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

General Terms Languages, Theory

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

1. Introduction

We have created a system for adding new type qualifiers to the Java language and enforcing their semantics at compile time. Our system provides a framework that extends existing Java APIs to facilitate compile time type-checking; it builds upon JSR 308 [3]. JSR 308 extends the syntax for Java annotations [1] so that they may be written anywhere that types are used, and extends the Java class file format so that these annotations are represented in the class file. The system includes features that are planned for inclusion in Java 7 (under JSR 308), and it is backward-compatible with Java 5.

In addition, we have developed type-checkers for several type qualifiers that are useful to programmers. One of these qualifiers, NonNull, provides an implicit subtype of each Java type that excludes the value null (i.e., a reference whose type is NonNull can never be null). Another, Interned, provides an implicit subtype denoting that a variable may be safely tested using the `==` operator (versus the `equals` method). A third [4] implements the Javari [5] language. A fourth implements the IGJ [6] language.

2. Motivation

Types help to detect and prevent errors by helping programmers to organize and document data, and by allowing tools like compilers to check that a program does not violate the type system's constraints. However, in languages like Java, there is often much information about a type that a programmer cannot express.

For instance, suppose a programmer wishes a variable to have the type "non-negative integer". The variable can be declared with the type `int`, but there is ordinarily no mechanism for expressing "non-negative". A custom type qualifier solves this problem: the programmer could use our tools to create a NonNegative qualifier and checker.

3. Implementation

Our system consists of two components.

The first component of the system, the extended annotations compiler, is an implementation of the JSR 308 specification [3]. It is composed of modifications to the `javac` Java compiler for parsing and compiling annotations on Java types. Java annotations are normally permitted on the declarations of classes, methods, and variables; the extended annotations compiler permits them anywhere types are written, including typecasts, type tests (`instanceof`), object creation expressions (`new`), method receivers, method `throws` clauses, generic type arguments, multidimensional arrays, type parameter bounds, and class literals. Moreover, the extended annotations compiler can write all annotations to the class file in a backward compatible way. This permits type-checking against binary versions of annotated classes

OOPSLA'07, October 21–25, 2007, Montréal, Québec, Canada.
ACM 978-1-59593-786-5/07/0010.

Cause	Errors
User type errors	3
User omissions	31
Run-time checks (application invariants)	23
Tool weaknesses	15
Incomplete flow-sensitivity	8
No qualified generic type inference	7

Table 1. Causes for errors issued by the NonNull type-checker plug-in during the case study.

(e.g., an annotated library) and will facilitate type-checking of Java bytecode. The extended annotations compiler also parses annotations written in C-style comments (`/* */`), so that code can be written for both the extended annotations compiler and older Java compilers.

The system's second component, the checkers framework, extends Java's annotation processing API [2] for compile-time type-checking of type qualifier annotations. The framework provides several features that reduce the time and effort required to create a new type-checker. First, it provides data structures for querying the annotations on a program element regardless of whether that element is found in a source file or in a class file. Second, it provides a template (using the visitor design pattern) for applying a type qualifier's rules to an input program, and it interfaces this component to the Java compiler. Third, the framework uses the Java compiler's messaging interface for reporting and collecting errors during type checking. Finally, the framework provides additional utilities for qualifiers that are either subtypes or supertypes of the unqualified type. For instance, NonNull and Interned types are subtypes of their unqualified types, and ReadOnly types are supertypes of their unqualified types.

Individual type-checkers may also include extra features beyond those provided by the checkers framework. The NonNull type-checker includes a flow-sensitive analysis that performs limited NonNull inference after explicit null checks.

4. Case study

In order to demonstrate that the current implementation is both usable and effective, we have conducted a case study in which we ran the NonNull checker on the NonNull-annotated source code for a library. The library, which provides routines for working with an "index file" that describes the type qualifiers in a class, consists of 4,640 lines of source code and contains 699 `@NonNull` annotations (out of 3,700 locations where a `@NonNull` annotation may have been written). The library author (who is not an author of this paper) added the annotations manually and without the aid of the NonNull type-checker (before the checker was written).

Table 1 shows the errors and warnings that resulted from running the type-checker on the annotated library. The first class of errors, user errors, are those for which the programmer's conception of a type was incorrect (i.e., thinking that a type was NonNull when it was possibly-null). These are serious errors that may lead to null-pointer exceptions at run time. The second class of errors, user omissions, are those in which the programmer forgot to write a `@NonNull` annotation for a NonNull type; they are easily fixed. The third class of errors, run-time checks, are those in which a possibly-null type but could be safely used as NonNull at certain locations where an application-specific invariant guarantees that the value is not null. We suppressed errors of this type by adding a run-time assertion (e.g., `assert x!=null;`) for each application invariant. The final class of errors, tool weaknesses, are those for which it was necessary to add an explicit null check to the subject code to satisfy the type-checker. We expect that improvements to the flow-sensitive analysis and inference for qualified generic types will eliminate this final class.

Using the framework described in Section 3, the NonNull type-checker consists of only 300 lines of Java source code (including comments), plus an additional 474 lines for the flow-sensitive analysis.

5. Downloads

The NonNull type-checker plug-in, the other three plug-ins mentioned in Section 1, the checkers framework, and the JSR 308 extended annotations Java compiler are publicly available for download from the JSR 308 web site, `http://pag.csail.mit.edu/jsr308`.

References

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

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

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

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

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

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