Preventing bugs with pluggable type checking

Michael D. Ernst
University of Washington
http://types.cs.washington.edu/jsr308
Motivation

java.lang.NullPointerException
Java’s type checking is too weak

• Type checking prevents many bugs
  ```java
  int i = "hello";   // type error
  ```

• Type checking doesn’t prevent **enough** bugs
  ```java
  System.console().readLine();
  ⇒ NullPointerException

  Collections.emptyList().add("One");
  ⇒ UnsupportedOperationException
  ```
Some errors are silent

```java
Date date = new Date(0);
myMap.put(date, "Java Epoch");
date.setYear(70);
myMap.put(date, "Linux Epoch");
⇒ Corrupted map

dbStatement.executeQuery(userInput);
⇒ SQL injection attack
```

Initialization, data formatting, equality tests, ...
Solution: Pluggable type systems

- Design a type system to solve a specific problem
- Write type qualifiers in code (or, type inference)
  ```java
  @Immutable Date date = new Date(0);
  date.setTime(70);    // compile-time error
  ```
- Type checker warns about violations (bugs)

% javac -processor NullnessChecker MyFile.java

MyFile.java:149: dereference of possibly-null reference bb2
  allVars = bb2.vars;
  ^
Benefits of pluggable types

- Improve documentation
- Find bugs in programs
- Guarantee the absence of errors
- Aid compilers, optimizers, and analysis tools
- Reduce number of assertions and run-time checks

Possible negatives:
- Must write the types (or use type inference)
- False positives are possible (can be suppressed)
Sample checkers

- Null dereferences
- Mutation and side-effects
- Equality tests
- Locking
- Security: encryption, tainting, access control
- Aliasing
- Strings: localization, regex syntax, ...
- Typestate (e.g., open/closed files)

- You can write your own!
Using checkers

• Designed as compiler plug-ins (i.e., annotation processors)
• Use familiar error messages

% javac -processor NullnessChecker MyFile.java

MyFile.java:9: incompatible types.
    nonNullVar = nullbleValue;
    ^
    found   : @Nullable String
required: @NonNull String
Demos available!

• Catch me anytime today or tomorrow
Checkers are effective

• Scales to > 200,000 LOC
• Each checker found errors in each code base it ran on
  – Verified by a human and fixed
## Comparison: other Nullness tools

<table>
<thead>
<tr>
<th>Checker Framework</th>
<th>Null pointer errors</th>
<th>False warnings</th>
<th>Annotations written</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Found</td>
<td>Missed</td>
<td></td>
</tr>
<tr>
<td>Checker Framework</td>
<td>8</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>FindBugs</td>
<td>0</td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>Jlint</td>
<td>0</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>PMD</td>
<td>0</td>
<td>8</td>
<td>0</td>
</tr>
</tbody>
</table>

- Checking the Lookup program for file system searching (4KLOC)
  - Distributed with Daikon (>100KLOC verified by our checker)
- False warnings are suppressed via an annotation or assertion
Checkers are featureful

- Full type systems: inheritance, overriding, etc.
- Generics (type polymorphism)
  - Also qualifier polymorphism
- Flow-sensitive type qualifier inference
- Qualifier defaults
- Warning suppression
Checkers are usable

• Integrated with toolchain
  – javac, Ant, Maven, Eclipse, Netbeans
• Few false positives
• Annotations are not too verbose
  – @NonNull: 1 per 75 lines
    • with program-wide defaults, 1 per 2000 lines
  – @Interned: 124 annotations in 220KLOC revealed 11 bugs
    – Possible to annotate part of program
    – Fewer annotations in new code
• Inference tools: nullness, mutability
What a checker guarantees

• The program satisfies the type property. There are:
  – no bugs (of particular varieties)
  – no wrong annotations

• Caveat 1: only for code that is checked
  – Native methods
  – Reflection
  – Code compiled without the pluggable type checker
  – Suppressed warnings
    • Indicates what code a human should analyze
  – Checking part of a program is still useful

• Caveat 2: The checker itself might contain an error
Annotating libraries

• Each checker comes with JDK annotations
  – Typically, only for signatures, not bodies
  – Finds errors in clients, but not in the library itself
• Inference tools for annotating new libraries
SQL injection attack

- Server code bug: SQL query constructed using unfiltered user input
  
  ```
  query = "SELECT * FROM users "
  + "WHERE name='" + userInput + 
  "';"
  ```

- User inputs: a' or 't'='t

- Result:
  
  ```
  query => SELECT * FROM users
  WHERE name='a' or 't'='t';
  ```

- Query returns information about all users
Taint checker

@TypeQualifier
@SubtypeOf(Unqualified.class)
@ImplicitFor(trees = {STRING_LITERAL})
public @interface Untainted { }

To use it:

1. Write `@Untainted` in your program
   
   ```java
   List getPosts(@Untainted String category) {...}
   ```

2. Compile your program
   
   ```bash
   javac -processor BasicChecker -Aquals=Untainted
   MyProgram.java
   ```
Defining a type system

@TypeQualifier
public @interface NonNull { }
Defining a type system

1. Type qualifier hierarchy
2. Type introduction rules
3. Other type rules

@TypeQualifier
public @interface NonNull { }
Defining a type system

1. Type qualifier hierarchy
2. Type introduction rules
3. Other type rules

```java
@TypeQualifier
@SubtypeOf( Nullable.class )
public @interface NonNull { }
```
Defining a type system

1. Type qualifier hierarchy
2. Type introduction rules
3. Other type rules

```java
@TypeQualifier
@SubtypeOf( Nullable.class )
@ImplicitFor(trees={ NEW_CLASS,
                    PLUS,
                    BOOLEAN_LITERAL, ... } )

public @interface NonNull { }
```
Defining a type system

1. Type qualifier hierarchy
2. Type introduction rules
3. Other type rules

```java
void visitSynchronized(SynchronizedTree node) {
    ExpressionTree expr = node.getExpression();
    AnnotatedTypeMirror type = getAnnotatedType(expr);
    if (!type.hasAnnotation(NONNULL)) {
        checker.report(Result.failure(...), expr);
    }
}
```
Pluggable type-checking

• Java 7 syntax for type annotations
  – Write in comments during transition to Java 7
• Checker Framework for creating type checkers
  – Featureful, effective, easy to use, scalable
• Prevent bugs at compile time
• Create custom type-checkers
• Learn more, or download the Checker Framework: http://types.cs.washington.edu/jsr308
  (or, web search for “Checker Framework” or “JSR 308”)