Preventing errors before they happen: Lightweight verification via pluggable type-checking

Michael D. Ernst
University of Washington (Seattle, WA, USA)
University of Buenos Aires
Joint work with Werner Dietl and many others

http://CheckerFramework.org/
Schedule

• Part 1 (11:00 – 12:30)
  – pluggable type-checking: what and why
  – demo of the Checker Framework
  – relevance to your programming problems

• Part 2 (14:00 – 15:30)
  – how to create your own type system
  – hands-on practice in using pluggable types
Motivation

Software bugs cost money
$312 billion per year (2013)
$440 million loss by Knight Capital Group in 30 minutes
$6 billion: 2003 blackout in northeastern USA & Canada

Software bugs cost lives
2003: 11 deaths: blackout
1997: 225 deaths: jet crash caused by radar software
1991: 28 deaths: Patriot missile guidance system
1985-2000: >8 deaths: Radiation therapy
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, ...

Goal: Find errors at compile time
... before testing, customers, or hackers find them
Solution: Pluggable type systems

• Design a type system to solve a specific problem
• Write type qualifiers in code (or, use type inference)

```java
@Immutable Date date = new Date(0);
date.setTime(70);    // compile-time error
```

• Type checker warns about violations (bugs)

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

MyFile.java:149: dereference of possibly-null reference bb2
    allVars = bb2.vars;
    ^
```
Outline

• Type qualifiers
• Pluggable type checkers
• Writing your own checker
• Verification vs. bug finding
• Conclusion
Type qualifiers

• **In Java 8**: annotations on types

```java
@Untainted String query;
List<@NonNull String> strings;
myGraph = (@Immutable Graph) tmpGraph;
@English String @ReadOnly [] words;
class UnmodifiableList<T>
    implements @Readonly List<@Readonly T> {}
```

• **Backward-compatible**: with any Java compiler

```java
List</*@NonNull*/ String> strings;
```
Benefits of type qualifiers

Find bugs in programs
• Guarantee the absence of errors

Improve documentation
• Improve code structure & maintainability
Aid compilers, optimizers, and analysis tools
• Reduce number of run-time checks

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

• Type qualifiers
• **Pluggable type checkers**
• Writing your own checker
• Verification vs. bug finding
• Conclusion
Using a checker

- Run in IDE or on command line
- Works as a compiler plug-in (annotation processor)
- Familiar workflow and error messages

% javac –processor NullnessChecker MyFile.java

MyFile.java:9: incompatible types.
nonNullVar = nullableValue;
^
found   : @Nullable String
required: @NonNull String

File.java

% java

0 errors, 1 warning, 0 others

Description

Warnings (1 item)

dereference of possibly-null reference console
Type Checking

Source → Compiler → Executable

No errors

Fix bugs
Change types

Errors
Optional Type Checking

- Source → Compiler → Executable
- Source → No errors → Compiler
- Compiler → Errors
- Errors → Fix bugs / Change types
- Errors → Warnings
- Warnings → Fix bugs
- Warnings → Add/change annotations
- Optional Type Checker → Guaranteed behavior
Optional Type Checking

Source → Compiler → Executable

- No errors
- Fix bugs
- Change types

Errors

- Fix bugs
- Add/change annotations

Optional Type Checker

Warnings

Guaranteed behavior
Nullness and mutation demo

• Detect errors
• Guarantee the absence of errors
• Verify the correctness of optimizations
Checkers are effective

**Practical:** in daily use at Google, on Wall Street, etc.

**Scalable:** > 6 MLOC checked at UW

Selected case study results:
- Signature strings: 28 errors in OpenJDK, ASM, AFU
- Nullness: >200 errors in Google Collections, javac, Daikon
- Interning: >200 problems in Xerces, Lucene
- Format strings: 104 errors, only 107 annotations required
- Regular expressions: 56 errors in Apache, etc.; 200 annos
- Fake enumerations: problems in Swing, JabRef
- Compiler messages: 8 wrong keys in Checker Framework
Comparison: other nullness tools

<table>
<thead>
<tr>
<th>Tool</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)
- False warnings are suppressed via an annotation or assertion
Checkers are featureful

- Full type systems: inheritance, overriding, generics (type polymorphism), etc.
- Type qualifier polymorphism
- Flow-sensitive type qualifier inference – no need to write annotations within method bodies
- Qualifier defaults
- Pre-/post-conditions, side effect annotations
- Warning suppression
Checkers are usable

- Integrated with toolchain
  - javac, Eclipse, Ant, Maven
- Annotations are not too verbose
  - @NonNull: 1 per 75 lines
    - with program-wide defaults, 1 per 2000 lines
  - @Interned: 124 annotations in 220 KLOC revealed 11 bugs
  - @Format: 107 annotations in 2.8 MLOC revealed 104 bugs
  - Possible to annotate part of program
  - Fewer annotations in new code

- Inference tools add annotations to your program
- Few false positives
- First-year CS majors preferred using checkers to not
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
Formalizations

\[
\begin{align*}
P & \in \text{Program} ::= \text{Class, ClassId, Expr}^r_T \\
\text{Cls} & \in \text{Class} ::= \text{class ClassId}<^\text{ClassId}<^\text{VarId} \\
& \quad \text{extends ClassId}<^\text{VarId}^r_T \{ \text{FieldId}<^\text{Type} \} \\
\& \quad \text{Meta} \\
\ast\text{T} & \in \ast\text{Type} ::= \ast\text{NType} | ^\text{VarId} \\
\ast\text{N} & \in \ast\text{NType} ::= \text{OM ClassId}<^\text{Type} \\
u & \in \text{OM} ::= \text{h, }^\text{r}\Gamma, e_0 \leadsto h_0, \nu_0 \\
mt & \in \text{Meth} ::= \nu_0 \neq \text{null}_a \\
\text{MethSig} & ::= h_0, ^\text{r}\Gamma, e_2 \leadsto h_2, \nu \\
w & \in \text{Purity} ::= \text{OS-Upd} \\
e & \in \text{Expr} ::= \text{h'} = h_2[\nu_0.f := \nu] \\
\ast\text{Gamma} & \in \ast\text{Env} ::= \\
\text{GT-Read} & \quad \Gamma \vdash e_0 : N_0 \\
& \quad \quad N_0 = u_0 \ C_0^{<>} \\
& \quad \quad \Gamma \vdash e_2 : N_0 \ D{T_1} \\
& \quad \quad u_0 \neq \text{any} \\
& \quad \quad \text{rp}(u_0, T_1) \\
\& \quad \quad \text{GT-Upd} \\
f & \in \text{Dyn} ::= \text{h, }^\text{r}\Gamma, e_0.f \leadsto h', \nu \\
\& \quad \quad \nu_1 : \text{dyn}(\ast\text{N}, h, \nu_1) \\
\& \quad \quad \nu_2 : \text{dyn}(\ast\text{T}, \nu_1, h(\nu_1)_1) \\
\& \quad \quad \ast\text{N} = u_N \ C_N^{<>} \\
u_N & = \text{this}_u \Rightarrow ^\text{r}\Gamma(\text{this}) \\
\& \quad \quad \text{free}(\ast\text{T}) \subseteq \text{dom}(C_N) \\
\& \quad \quad \text{Dyn} \\dysn & \in \text{Dyn} ::= \text{h, }^\text{r}\Gamma, (X' \text{r}_T^{'}, -) = \ast\text{T}[\nu'/\text{this}, \nu'/\text{peer}, \nu/\text{rep}, \text{any}_a/\text{any}_u, \text{r}_T/X, \text{r}_T'/X']
\end{align*}
\]
Annotating libraries

• Each checker comes with JDK annotations
  – For signatures, not bodies
  – Finds errors in clients, but not in the library itself
• Inference tools for annotating new libraries
What bugs can you detect & prevent?

The property you care about:
Null dereferences
Mutation and side-effects
Concurrency: locking
Security: encryption, tainting
Aliasing
Equality tests
Strings: localization, regular expression syntax, signature representation, format string syntax
Enumerations
Typestate (e.g., open/closed files)

The annotation you write:
@NonNull
@Immutable
@GuardedBy
@Encrypted
@OsTrusted, @Untaint...
@Linear
@Interned
@Localized
@Regex
@FullyQualified
@Format
@Fenum
@State

Users can write their own checkers!
Outline

• Type qualifiers
• Pluggable type checkers
• **Writing your own checker**
• Verification vs. bug finding
• Conclusion
Example: Regular expressions

// Prints the first matching group.
// For example:
//   java RegexExample ([0-9]*):([0-9]*) 23:59
// prints "Group 1 = 23"

public static void main(String[] args) {
    String regex = args[0];
    String content = args[1];
    Pattern pat = Pattern.compile(regex);
    Matcher mat = pat.matcher(content);
    if (mat.matches()) {
        System.out.println("Group 1 = " + mat.group(1));
    }
}
Regular expression type system

- What runtime errors to prevent? 
  PatternSyntaxException and IndexOutOfBoundsException.

- What operations are legal? 
  Pattern.compile only on valid regex. 
  Matcher.group(i) only if >i groups.

- What properties of data should hold? 
  Strings: valid regex vs. invalid. 
  Number of groups in a regex.
Example: Encrypted communication

```java
void send(@Encrypted String msg) {...}

@Encrypted String msg1 = ...;
send(msg1); // OK

String msg2 = ...;
send(msg2); // Warning!
```
Encryption type system

- What runtime exceptions to prevent? 
  Invalid information flow.
- What operations are legal? 
  send() only on encrypted data.
- What properties of data should hold? 
  Separate encrypted from plaintext strings.
Brainstorming new type checkers

- What runtime exceptions to prevent?
- What operations are legal and illegal?
- What properties of data should hold?

- Type-system checkable properties:
  - Dependency on values
  - Not on program structure, timing, ...
Brainstorming

- What runtime exceptions to prevent?
- What operations are legal and illegal?
- What properties of data should hold?
Outline

• Type qualifiers
• Pluggable type checkers
• Writing your own checker
• Verification vs. bug finding
• Conclusion
SQL injection attack

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

- User inputs: a’ or ‘1’=‘1

- Result:
  
  ```sql
  query ⇒ SELECT * FROM users
  WHERE name='a' or '1'='1';
  ```

- 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
   
   List getPosts(@Untainted String category) {...}

2. Compile your program
   
   javac -processor BasicChecker -Aquals=Untainted MyProgram.java
Taint checker demo

• Detect SQL injection vulnerability
• Guarantee absence of such vulnerabilities
Defining a type system

@TypeQualifier
public @interface NonNull { }

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

1. Qualifier hierarchy – rules for assignment
2. Type introduction – types for expressions
3. Type rules – checker-specific errors

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

1. Qualifier hierarchy
2. Type introduction
3. Type rules

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

1. Qualifier hierarchy
2. Type introduction
3. Type rules

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

public @interface NonNull { }

Gives the type of expressions:

new Date()
"hello " + getName()
Boolean.TRUE
Defining a type system

1. Qualifier hierarchy
2. Type introduction
3. Type rules

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

Errors for unsafe code:

synchronized (expr) {
    ...
}

Warn if expr may be null
Outline

- Type qualifiers
- Pluggable type checkers
- Writing your own checker
- Verification vs. bug finding
- Conclusion
Verification

• **Goal**: prove that no bug exists
• **Specifications**: user provides
• **False negatives**: none
• **False positives**: user suppresses warnings
• **Downside**: user burden

Bug-finding

• **Goal**: find some bugs at low cost
• **Specifications**: infer likely specs
• **False negatives**: acceptable
• **False positives**: heuristics focus on most important bugs
• **Downside**: missed bugs

Neither is “better”; each is appropriate in certain circumstances. The approaches are converging.
Other design considerations

- Visibility of specifications and warning suppressions
  - In the source code
    - documentation aids programmer understanding
  - In the tool
    - reduces code clutter

- Analysis comprehensibility
  - A transparent tool gives understandable outcomes
    - requires more upfront effort; more false positives
  - An opaque tool can use more powerful analyses
    - requires more effort to understand warnings
Outline

• Type qualifiers
• Pluggable type checkers
• Writing your own checker
• Verification vs. bug finding
• Hands-on practice
• Conclusion
How to get started

1. Write the specification
   Search the Javadoc for occurrences of “null”
   Replace the wordy English text by \texttt{@Nullable}
   Can also search code, but no annos in methods

2. Run Nullness Checker: verify/improve spec
   For each warning:
   – Reason about whether the code is safe
   – Express that reasoning as annotations
   – Consider improving the code’s design
Tips

What to type-check:

• Only type-check properties that matter to you
  – Use subclasses (not type qualifiers) if possible
• Choose part of your code to type-check first
  – Eliminate raw types such as List; use List<String>

While you are doing type-checking:

• Write the spec first (and think of it as a spec)
• Avoid warning suppressions when possible
Your turn to improve your code!

1. Choose a project you care about
2. Improve it
   - Apply an existing checker to your code, or
   - Create a new domain-specific type checker

Or, try the tutorial:
http://types.cs.washington.edu/checker-framework/tutorial
Outline

- Type qualifiers
- Pluggable type checkers
- Writing your own checker
- Verification vs. bug finding
- Conclusion
Pluggable type-checking

- Java 8 syntax for type annotations
- **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:
  (or, web search for “Checker Framework”)