void print(@Readonly Object x) {
    List<@NonNull String> lst;
    ...
}

Practical pluggable types via the Checker Framework

Matthew Papi, Mahmood Ali, Telmo Correa Jr., Jeff Perkins, Michael Ernst
MIT
Problem 1: Java’s type checking

- Type checking prevents many bugs
  ```java
  int i = "hello"; // type error
  ```
- Type checking doesn’t prevent **enough** bugs
  ```java
  java.lang.NullPointerException
  at checkers.util.GraphQualifierHierarchy.isSubtype(GraphQualifierHierarchy.java:21)
  at checkers.types.QualifierHierarchy.isSubtype(QualifierHierarchy.java:70)
  at checkers.types.TypeHierarchy.isSubtypeImpl(TypeHierarchy.java:100)
  at checkers.types.TypeHierarchy.isSubtype(TypeHierarchy.java:64)
  at checkers.basetype.BaseTypeChecker.isSubtype(BaseTypeChecker.java:305)
  at checkers.basetype.BaseTypeVisitor.commonAssignmentCheck(BaseTypeVisitor.java:48)
  at checkers.basetype.BaseTypeVisitor.commonAssignmentCheck(BaseTypeVisitor.java:46)
  at checkers.basetype.BaseTypeVisitor.checkArguments(BaseTypeVisitor.java:597)
  at checkers.basetype.BaseTypeVisitor.visitMethodInvocation(BaseTypeVisitor.java:27)
  at com.sun.tools.javac.tree.JCTree$JCMethodInvocation.accept(JCTree.java:1315)
  at com.sun.source.util.TreePathScanner.scan(TreePathScanner.java:239)
  at com.sun.tools.javac.tree.JCTree$JCExpressionStatement.accept(JCTree.java:1155)
  at com.sun.source.util.TreePathScanner.scan(TreePathScanner.java:67)
  at com.sun.source.util.TreeScanner.scanAndReduce(TreeScanner.java:80)
  ```
Problem 1: Java’s type checking

• Type checking prevents many bugs
  \[
  \text{int } i = "hello"; \quad // \text{ type error}
  \]

• Type checking doesn’t prevent enough bugs
  – Null dereferences
  – Incorrect equality tests
  – Incorrect mutation
  – SQL injection
  – Privacy violations
  – Misformatted data
  – ...
Solution: Pluggable type systems

• Design a type system to solve a specific problem
• Write type qualifiers in your code (or, type inference)
  ```java
  @NonNull Date d = ...;
  d.getMonth();    // no possible NPE
  
  • Type checker warns about violations (bugs)
  ```

% javac -processor NullnessChecker MyFile.java

MyFile.java:149: dereference of possibly-null reference bb2
  if (vars1.containsAll(bb2.vars))
    ^
Problem 2: Implementing pluggable types

- Modify a compiler: hard
- Previous frameworks: too weak
- Alternatives to case studies:
  - Soundness proof for core calculus
  - Toy examples
- Hampers understanding and use of type systems
- A type system is valuable only if it helps developers to find and prevent errors
  - Case studies are necessary
Example:
Type systems for immutability

```java
println(@Readonly Object x) { ... }

@Readonly Object[] getSigners() { ... }

Map<@Immutable Date, Object> cache;
```

- Formalisms, proofs, etc.
- Crucial insight from case studies
  - Javari type system [Birka 2004]: 160 KLOC
  - IGJ type system [Zibin 2007]: 106 KLOC
- Larger case studies have revealed even more
Solution: The Checker Framework

- Enables creation of **pluggable types** for Java
- **Expressive**
  - Can create realistic type systems
- **Concise**
  - Most checks are built in
  - Declarative syntax for common cases
- **Scalable**
  - Large programs
  - Full Java language
  - Java toolchain
- Aids **programmers and type system designers**
Contributions

• Syntax for type qualifiers in Java
• Checker Framework for writing type checkers
• 5 checkers written using the framework
• Case studies enabled by the infrastructure
• Insights about the type systems
Syntax for type qualifiers

- **Java 7**: annotate any use of a type
  
  ```java
  List<@NonNull String> myStrings;
  myGraph = (@Immutable Graph) tmpGraph;
  class UnmodifiableList<T>
      implements @Readonly List<@Readonly T> {}
  ```

- **Backward-compatible**: compile with any Java compiler
  
  ```java
  List</*@NonNull*/ String> myStrings;
  ```
Tool integration

- IDE support: javac, Eclipse, Netbeans
  - IntelliJ planned
- Annotated JDK
- Type inference: 3 tools
- Building type checkers:

- Integrating with other tools:
  - IBM, INRIA, JetBrains, MIT, Oxford, Sun, Victoria University of Wellington, ...
Outline

• Syntax for type qualifiers in Java
• **Checker Framework** for writing type checkers
• 5 checkers written using the framework
• Case studies enabled by the infrastructure
• Insights about the type systems
• Conclusion
A complete, useful type checker

@TypeQualifier
@SubtypeOf(Unqualified.class)
public @interface Encrypted {}

To use it:

1. Write @Encrypted in your program
   
   ```java
   void send(@Encrypted String message) { ... }
   ```

2. Compile your program

   ```bash
   javac -processor BasicChecker -Aquals=Encrypted MyProgram.java
   ```
Built-in features

• Arbitrary type qualifier hierarchy
• Subtyping: inheritance, overriding, assignment
• Polymorphism
  – Types (Java generics)
  – Type qualifiers
• Type qualifier inference (flow-sensitivity)
• Implicit and default qualifiers
• ...

Defining a type system

@TypeQualifier
public @interfaceNonNull { }
Defining a type system

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

@TypeQualifier
class NonNull { }

@NonNull
class NonNull { }
Defining a type system

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

@TypeQualifier
@SubtypeOf( Nullable.class )

public @interface NonNull { }
Defining a type system

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

```
new Date()
“hello ” + getName()
Boolean.TRUE
```

```
@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);
}
```

`synchronized(expr) {
  ...
}

Warn if expr may be null
Type refinement

Date d1, d2, d3;   // may be null

d1 = new Date();
d1.getMonth();     // OK: d1 is non-null

assert d2 != null;
d2.getMonth();     // OK

if (d3 != null) {
    d3.getMonth();   // OK
}

Type-checks as if annotations/casts were present
“Local flow-sensitive type qualifier inference”
Outline

• Syntax for type qualifiers in Java
• Checker Framework for writing type checkers
• 5 checkers written using the framework
• Case studies enabled by the infrastructure
• Insights about the type systems
• Conclusion
Sample type checkers

- Basic checker (subtyping)
- Null dereferences (@NonNull)
- Errors in equality testing (@Interned)
- Reference immutability (Javari)
- Reference & object immutability (IGJ)

< 500 LOC per checker
Outline

• Syntax for type qualifiers in Java
• Checker Framework for writing type checkers
• 5 checkers written using the framework
• Case studies enabled by the infrastructure
• Insights about the type systems
• Conclusion
Case studies

• Annotated existing Java programs
• Found bugs in every codebase
  – Verified by a human and fixed
• As of summer 2007: 360 KLOC
  – Now: > 1 MLOC
  – Scales to > 200 KLOC
Checkers are easy to use

- Natural part of workflow
- Feels like Java
- Few false positives
  - Easy to understand and fix
Annotations are not too verbose

• Examples:
  – Nullness: 1 per 75 lines
  – Interning: 124 in 220 KLOC revealed 11 bugs
• Careful choice of defaults
• Type refinement
• Possible to annotate part of program
• Fewer annotations in new code
## Comparison: other Nullness tools

<table>
<thead>
<tr>
<th></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 a 4KLOC program
- The other tools find bugs besides null dereferences
Outline

• Syntax for type qualifiers in Java
• Checker Framework for writing type checkers
• 5 checkers written using the framework
• Case studies enabled by the infrastructure
• Insights about the type systems
• Conclusion
Lessons learned

• Type systems
  – Interning
  – Nullness
  – Javari
  – IGJ

• Polymorphism

• Framework design

• Others: supertype qualifiers, simple type systems, inference, syntax, language integration, toolchain, ...

Interning type system

• New approach to finding equality errors
  – Purely type system
  – Fully backward compatible
  – Permits both interned and uninterned instances
Nullness type system

• New default: non-null except locals (NNEL)
  • Signatures: non-null
  • Locals: nullable
    – Inspired by practical use
    – Exploits flow sensitivity
    – Applicable to other type systems

• Nullness differs from other type systems
  – More application invariants
  – Run-time checks
  – Needs flow-sensitivity

<table>
<thead>
<tr>
<th>Default</th>
<th># annotations</th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>@Nullable</td>
<td>382</td>
<td>11</td>
</tr>
<tr>
<td>@NonNull</td>
<td>80</td>
<td>2.3</td>
</tr>
<tr>
<td>NNEL</td>
<td>35</td>
<td>1.0</td>
</tr>
</tbody>
</table>
Javari type system

• Enhancements:
  – Permit covariant method parameters
    ```java
class Super { void mymethod(Object x); }
class Sub   { void mymethod(@Readonly Object x); }
```
  – Improved type parameter inference
  – Improved treatment of fields
IGJ type system

• Need class, object, and reference immutability
• IGJ weakness: visitor pattern and callbacks
• In SVNKit:
  – Getters have side effects
  – Setters have no side effects
Polymorphism

• Multiple varieties:
  – Type polymorphism (Java generics including wildcards)
  – Qualifier polymorphism (different context sensitivity)
  – Containing-object context (deep immutability)

• Qualifier polymorphism: one qualifier is enough
  – Linear-time inference (with flow- & context-sensitivity)

• Polymorphism dominated all other problems
  – Type system, design, and implementation challenges
  – Evaluation cannot ignore generics
Framework design

• Checker and compiler are decoupled
• All type rules are written in Java
  – Integrated declarative and procedural mechanisms
• Procedural code is necessary
  – Don’t automate the type systems of yesterday
• Representation of annotated types
• Framework is good for more than type checking
Outline

• Syntax for type qualifiers in Java
• Checker Framework for writing type checkers
• 5 checkers written using the framework
• Case studies enabled by the infrastructure
• Insights about the type systems
• Conclusion
Related work

• JavaCOP [Andreae 2006]
  – Pattern-matching syntax, requires Java helper methods
  – No Java generics, not scalable, no realistic evaluation yet

• JQual [Greenfieldboyce 2007]:
  – Inference framework, same basic rules as [Tschantz 2006]
  – No Java generics, limited expressiveness, no checker, not precise, not sound

• JavaCOP and JQual authors attempted to implement Javari
  – Only 1 out of 5 keywords, incorrect method overriding

• JastAdd [Ekman 2007]
  – Extensible compiler framework
  – Not concise
Contributions

• Checker Framework for creating type checkers
  – Robust, scalable, easy to use
  – Demos available
  – http://pag.csail.mit.edu/jsr308

• For programmers: prevents errors

• For type system designers:
  – Enables construction & evaluation of type systems
  – Yields insight into type systems