

Preventing Runtime Errors at Compile Time using the *Checker Framework*



<http://CheckerFramework.org/>

@CheckerFrmwrk on Twitter
CheckerFramework on Facebook & Google+



Werner Dietl

University of Waterloo

<https://ece.uwaterloo.ca/~wdietl/>



Joint work with Michael D. Ernst and many others.

Bug Evolution

Photo # NH 96566-KN (Color) First Computer "Bug", 1947


9/2
9/9

0800 Antman started
1000 " stopped - antman ✓

1300 (032) MP - MC 1.582640000
(033) PRO 2 2.130476415
convd 2.130676415

Relays 6-2 in 033 failed special speed test
in relay 10,000 test.

1100 Started Cosine Tape (Sine check)
1525 Started Multi-Adder Test.

1545  Relay #70 Panel F
(moth) in relay.

First actual case of bug being found.

1630 Antman started.
1700 closed down.

Relay 3145
Relay 3270

<http://www.history.navy.mil/photos/images/h96000/h96566k.jpg>



<http://checkerframework.org/>

Bug Evolution

Photo # NH 96566-KN (Color) First Com

9/2
9/9
0800 Archan started
1000 " stopped - archan ✓
1300 (032) MP - MC 2.130
(033) PRO 2 2.130
convd 2.130
Relays 6-2 in 033 failed
in relay
Relays changed
1100 Started Cosine Tape (Sin
1525 Started Multi-Adder To
1545 Relays (moth)
First actual case of bug
1630 Archan started.
1700 closed down.



<http://dimages.businessweek.com/imageserve/0arRbTIdRSg3e/630x418.jpg>



<http://checkerframework.org/>

Cost of software failures

\$312 billion per year global cost of software bugs (2013)

\$300 billion dealing with the Y2K problem

\$440 million loss by Knight Capital Group Inc. in 30 minutes in August 2012

\$650 million loss by NASA Mars missions in 1999; unit conversion bug

\$500 million Ariane 5 maiden flight in 1996; 64 bit to 16 bit conversion bug



Software bugs can cost lives

225 deaths: jet crash caused by radar software (1997)

28 deaths: Patriot missile guidance system (1991)

11 deaths: blackout (2003)

>8 deaths: Radiation therapy (1985-2000)

2011: Software cause for 25% of all medical device recalls



Outline

- Solution: Pluggable type-checking
- Tool: Checker Framework
- Experience report
- Creating your own type system
- Java 8 type annotation features
- Preventing Null-Pointer Exceptions



Java's type system is too weak

Type checking prevents many errors

```
int i = "hello";
```

Type checking doesn't prevent **enough** errors

```
System.console().readLine();
```

```
Collections.emptyList().add("one");
```



Java's type system is too weak

Type checking prevents many errors

```
int i = "hello";
```

Type checking doesn't prevent **enough** errors

```
System.console().readLine();
```

```
Collections.  
NullPointerException
```



Java's type system is too weak

Type checking prevents many errors

```
int i = "hello";
```

Type checking doesn't prevent **enough** errors

```
System.console().readLine();
```

```
Collections.emptyList().add("one");
```



Java's type system is too weak

Type checking prevents many errors

```
int i = "hello";
```

Type checking doesn't prevent **enough** errors

```
System.console().readLine();
```

```
Collections.emptyList().add("one");
```

UnsupportedOperationException



Some errors are silent

```
Date date = new Date();  
myMap.put(date, "now");  
date.setSeconds(0);    // round to minute  
myMap.get(date);
```



Some errors are silent

```
Date date = new Date();  
myMap.put(date, "now");  
date.setSeconds(0); // round to minute  
myMap.get(date);
```



Element not found



Some errors are silent

```
dbStatement.executeQuery(userInput);
```



Some errors are silent

```
dbStatement.executeQuery(userInput);
```

SQL injection attack

Initialization, data formatting, equality tests, ...



Solution: Pluggable Type Checking

1. Design a type system to solve a specific problem
2. Write type qualifiers in code (or, use type inference)
3. Type checker warns about violations (bugs)



Solution: Pluggable Type Checking

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

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

3. Type checker warns about violations (bugs)



Solution: Pluggable Type Checking

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

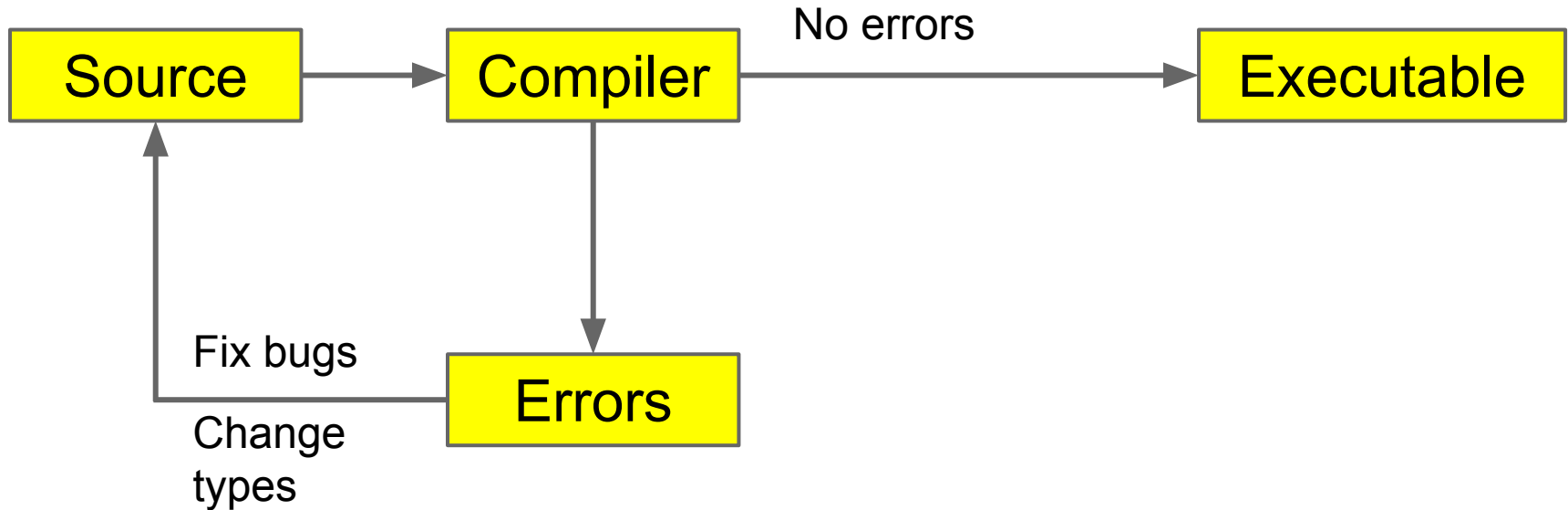
```
@Immutable Date date = new Date();  
date.setSeconds(0); // compile-time error
```

3. Type checker warns about violations (bugs)

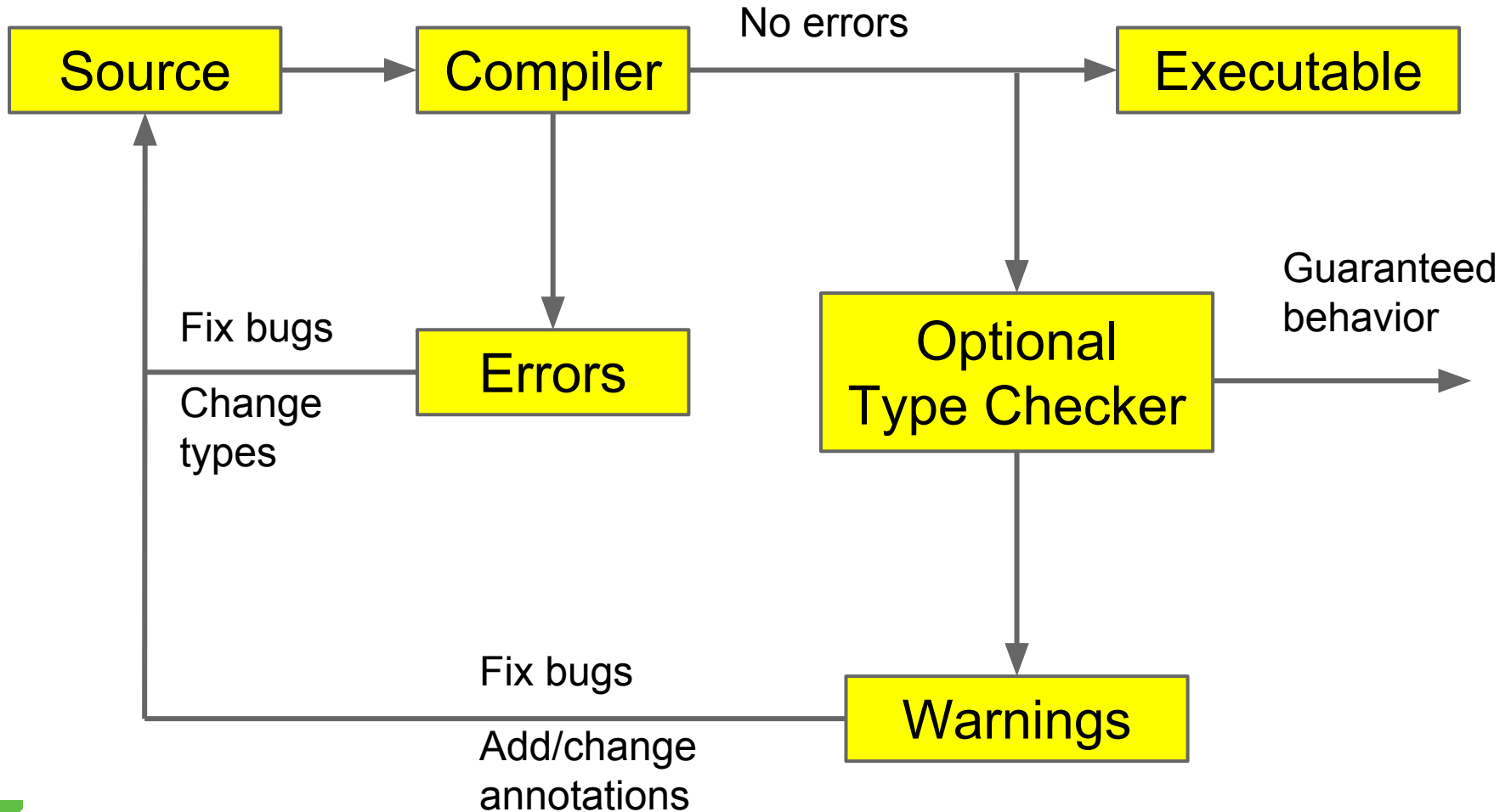
```
% javac -processor NullnessChecker MyFile.java  
MyFile.java:149: dereference of possibly-null  
reference bb2  
    allVars = bb2.vars;  
                ^
```



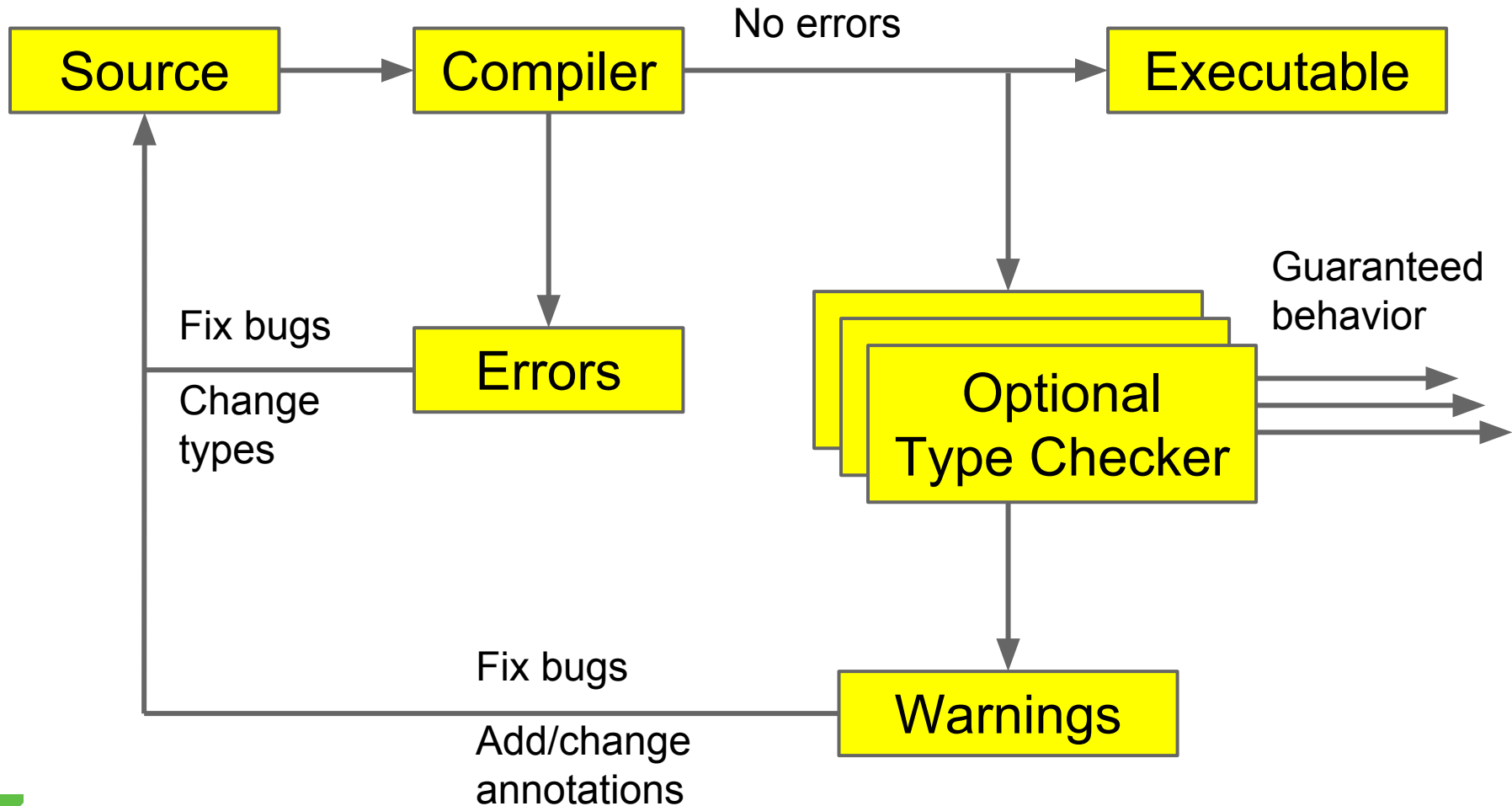
Type Checking



Optional Type Checking



Optional Type Checking



Prevent null pointer exceptions

Type system that statically guarantees that the program only dereferences known non-null references

Types of data

@NonNull reference is never null

@Nullable reference may be null



Null pointer exception

```
String op(Data in) {  
    return "transform: " + in.getF();  
}
```

...

```
String s = op(null);
```



Null pointer exception

Where is the defect?

```
String op(Data in) {  
    return "transform: " + in.getF();  
}
```

...

```
String s = op(null);
```



Null pointer exception

Where is the defect?

```
String op(Data in) {  
    return "transform: " + in.getF();  
}
```

...

```
String s = op(null);
```



Null pointer exception

Where is the defect?

```
String op(Data in) {  
    return "transform: " + in.getF();  
}
```

...

```
String s = op(null);
```

Can't decide without specification!



Specification 1: non-null parameter

```
String op(@Nonnull Data in) {  
    return "transform: " + in.getF();  
}
```

...

```
String s = op(null);
```



Specification 1: non-null parameter

```
String op(@NonNull Data in) {  
    return "transform: " + in.getF();  
}
```

...

```
String s = op(null);           // error
```



Specification 2: nullable parameter

```
String op(@Nullable Data in) {  
    return "transform: " + in.getF();  
}
```

...

```
String s = op(null);
```



Specification 2: nullable parameter

```
String op(@Nullable Data in) {  
    return "transform: " + in.getF();  
}                                     // error
```

...

```
String s = op(null);
```



Benefits of type systems

- **Find bugs** in programs
 - Guarantee the **absence of errors**
- **Improve documentation**
 - Improve code structure & maintainability
- Aid compilers, optimizers, and analysis tools
 - E.g., could reduce number of run-time checks



Benefits of type systems

- **Find bugs** in programs
 - Guarantee the **absence of errors**
- **Improve documentation**
 - Improve code structure & maintainability
- Aid compilers, optimizers, and analysis tools
 - E.g., could reduce number of run-time checks
- Possible negatives:
 - Must write the types (or use type inference)
 - False positives are possible (can be suppressed)



Input Format Validation

Demo: ensure that certain strings contain **valid regular expressions**.



Regular Expression Example

```
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: " + mat.group(1));  
    }  
}
```



Regular Expression Example

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

PatternSyntaxException

IndexOutOfBoundsException



Fixing the Errors

`Pattern.compile` only on valid regex

`Matcher.group(i)` only if $> i$ groups

...

```
if (!RegexUtil.isRegex(regex, 1)) {  
    System.out.println("Invalid: " + regex);  
    System.exit(1);  
}
```

...



The Checker Framework

A framework for pluggable type checkers
“Plugs” into the OpenJDK or OracleJDK
compiler

```
javac -processor MyChecker ...
```

Standard error format allows tool integration



Eclipse plug-in

```
3 public class Test {  
4  
5     public static void main(String[] args) {  
6         Console c = System.console();  
7         c.printf("Test");  
8     }  
9 }
```

Problems @ Javadoc Declaration Search

0 errors, 1 warning, 0 others

Description

Warnings (1 item)

dereference of possibly-null reference c
c.printf("Test");

```
3 public class Test {  
4  
5     public static void main(String[] args) {  
6         Console c = System.console();  
7         dereference of possibly-null reference c c.printf("Test");  
8     }  
9 }
```

Problems @ Javadoc Declaration Search Console Task

0 errors, 1 warning, 0 others

Description	Resource
Warnings (1 item)	
dereference of possibly-null reference c c.printf("Test");	Test.java



Ant and Maven integration

```
<presetdef name="jsr308.javac">
  <javac fork="yes"
    executable="${checkerframework}/checker/bin/${cfJavac}" >
    <!-- JSR-308-related compiler arguments -->
    <compilerarg value="-version"/>
    <compilerarg value="-implicit:class"/>
  </javac>
</presetdef>
```

```
<dependencies>
  ... existing <dependency> items ...
  <!-- annotations from the Checker Framework:
    nullness, interning, locking, ... -->
  <dependency>
    <groupId>org.checkerframework</groupId>
    <artifactId>checker-qual</artifactId>
    <version>1.9.7</version>
  </dependency>
</dependencies>
```



Web interface

<http://eisop.uwaterloo.ca/live/>

Checker Framework Live Demo

Write Java code here:

```
1 import org.checkerframework.checker.nullness.qual.Nullable;
2 class YourClassNameHere {
3     void foo(Object nn, @Nullable Object nbl) {
4         nn.toString(); // OK
5         nbl.toString(); // Error
6     }
7 }
```

Choose a type system:

Check

Examples:

Nullness: [NullnessExample](#) | [NullnessExampleWithWarnings](#)

MapKey: [MapKeyExampleWithWarnings](#)

Interning: [InterningExample](#) | [InterningExampleWithWarnings](#)

Lock: [GuardedByExampleWithWarnings](#) | [HoldingExampleWithWarnings](#) | [EnsuresLockHeldExample](#) | [Local](#)

Example type systems

Null dereferences (`@NonNull`)

>200 errors in Google Collections, javac, ...

Equality tests (`@Interned`)

>200 problems in Xerces, Lucene, ...

Concurrency / locking (`@GuardedBy`)

>500 errors in BitcoinJ, Derby, Guava, Tomcat, ...

Fake enumerations (`@Fenum`)

problems in Swing, JabRef



String type systems

Regular expression syntax (`@Regex`)

56 errors in Apache, etc.; 200 annos

printf format strings (`@Format`)

104 errors, only 107 annotations required

Signature format (`@FullyQualified`)

28 errors in OpenJDK, ASM, AFU

Compiler messages (`@CompilerMessageKey`)

8 wrong keys in Checker Framework



Security type systems

Command injection vulnerabilities (`@OsTrusted`)
5 missing validations in Hadoop

Privacy (`@Source`, `@Sink`)
SPARTA detected malware in Android apps



You can write your own checker!



Brainstorming new type checkers

What runtime exceptions to prevent?

What properties of data should always hold?

What operations are legal and illegal?

Type-system checkable properties:

- Dependency on values
- Not on program structure, timing, ...



Checkers are usable

- Type-checking is **familiar** to programmers
- Modular: fast, incremental, partial programs
- Annotations are **not too verbose**
 - **@NonNull**: 1 per 75 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
- Few false positives
- First-year CS majors preferred using checkers to not
- **Practical**: in daily use at Google, on Wall Street, etc.



Comparison: other nullness tools

	Null pointer errors		False warnings	Annotations written
	Found	Missed		
Checker Framework	8	0	4	35
FindBugs	0	8	1	0
Jlint	0	8	8	0
PMD	0	8	0	0

Checking the Lookup program for file system searching (4kLOC)

False warnings are suppressed via an annotation or assertion



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 (but handles 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

<h1>Formalizations</h1>		$h \in \text{Heap}$	$= \text{Addr} \rightarrow \text{Obj}$
		$\iota \in \text{Addr}$	$= \text{Set of Addresses} \cup \{\text{null}_a\}$
		$o \in \text{Obj}$	$= {}^r\text{Type}, \text{Fields}$
$P \in \text{Program}$	$::= \overline{\text{Class}}, \text{ClassId}, \text{Expr}$	${}^rT \in {}^r\text{Type}$	$= \text{OwnerAddr ClassId} \langle \overline{{}^r\text{Type}} \rangle$
$\text{Cls} \in \text{Class}$	$::= \text{class ClassId} \langle \overline{{}^s\text{TVarId}} \rangle$	$\text{Fs} \in \text{Fields}$	$= \text{FieldId} \rightarrow \text{Addr}$
	$\text{extends ClassId} \langle \overline{{}^s\text{Type}} \rangle$	$\iota \in \text{OwnerAddr}$	$= \text{Addr} \cup \{\text{any}_a\}$
	$\{ \text{FieldId } {}^s\text{Type}; \text{Met}$	${}^r\Gamma \in {}^r\text{Env}$	$= \overline{\text{TVarId } {}^r\text{Type}; \text{ParId Addr}}$
${}^sT \in {}^s\text{Type}$	$::= {}^s\text{NType} \mid \text{TVarId}$		
${}^sN \in {}^s\text{NType}$	$::= \text{OM ClassId} \langle \overline{{}^s\text{Type}} \rangle$		
$u \in \text{OM}$	$::=$	$h, {}^r\Gamma, e_0 \rightsquigarrow h_0, \iota_0$	$h, {}^r\Gamma, e_0 \rightsquigarrow h', \iota_0$
$\text{mt} \in \text{Meth}$	$::=$	$\iota_0 \neq \text{null}_a$	$\iota_0 \neq \text{null}_a$
$\text{MethSig} \in$	$::=$	$h_0, {}^r\Gamma, e_2 \rightsquigarrow h_2, \iota$	$\text{OS-Read} \frac{\iota = h'(\iota_0) \downarrow_2 (f)}{h, {}^r\Gamma, e_0.f \rightsquigarrow h', \iota}$
$w \in \text{Purity}$	$::=$	$h' = h_2[\iota_0.f := \iota]$	
$e \in \text{Expr}$	$::=$	$\text{OS-Upd} \frac{h, {}^r\Gamma, e_0.f = e_2 \rightsquigarrow h'}{h, {}^r\Gamma, e_0.f = e_2 \rightsquigarrow h'}$	
${}^s\Gamma \in {}^s\text{Env}$	$::=$	$\text{Expr.MethId} \langle \overline{{}^s\text{Type}} \rangle (\text{Expr}) \mid$	$\Gamma \vdash e_0 : N_0 \quad N_0 = u_0 C_0 \langle _ \rangle$
		$\text{new } {}^s\text{Type} \mid ({}^s\text{Type}) \text{ Expr}$	$T_1 = fType(C_0, f)$
		$\overline{\text{TVarId } {}^s\text{NType}; \text{ParId } {}^s\text{Type}}$	$\Gamma \vdash e_2 : N_0 \triangleright T_1$
			$\text{GT-Read} \frac{\Gamma \vdash e_0 : N_0 \quad N_0 = _}{\Gamma \vdash e_0.f : N_0 \triangleright fType(C_0, f)}$
$h \vdash {}^r\Gamma : {}^s\Gamma$			$\text{GT-Upd} \frac{u_0 \neq \text{any} \quad rp(u_0, T_1)}{\Gamma \vdash e_0.f = e_2 : N_0 \triangleright T_1}$
$h \vdash \iota_1 : \text{dyn}({}^sN, h, \iota_1)$			
$h \vdash \iota_2 : \text{dyn}({}^sT, \iota_1, h(\iota_1) \downarrow_1)$			
${}^sN = u_N C_N \langle _ \rangle$			
$u_N = \text{this}_u \Rightarrow {}^r\Gamma(\text{this})$			
$\text{free}({}^sT) \subseteq \text{dom}(C_N)$			
$\text{DYN} \frac{\left\{ \begin{array}{l} \implies h \vdash \iota_2 : \text{dyn}({}^sN \triangleright {}^sT, h, {}^r\Gamma) \\ {}^rT = \iota' \langle _ \rangle \quad \iota \vdash {}^rT \prec: \iota' C \langle \overline{{}^rT} \rangle \quad \iota \vdash {}^rT \prec: \iota' C \langle \overline{{}^rT}_a \rangle \Rightarrow \iota \vdash \overline{{}^rT} \prec: \overline{{}^rT}_a \\ \text{dom}(C) = \overline{X} \quad \text{free}({}^sT) \subseteq \overline{X} \circ \overline{X}' \end{array} \right.}{\text{dyn}({}^sT, \iota, {}^rT, (\overline{X}' \overline{{}^rT}'; _)) = {}^sT[\iota'/\text{this}, \iota'/\text{peer}, \iota/\text{rep}, \text{any}_a/\text{any}_u, \overline{{}^rT}/\overline{X}, \overline{{}^rT}'/\overline{X}']}$			



Since Java 5: declaration annotations

Only for declaration locations:

@Deprecated

```
class Foo {  
    @Getter @Setter private String query;  
    @SuppressWarnings("unchecked")  
    void foo() { ... }  
}
```



But we couldn't express

A non-null reference to my data

An interned String

A non-null List of English Strings

A non-empty array of English strings



With Java 8 Type Annotations we can!

A non-null reference to my data

```
@NonNull Data mydata;
```

An interned String

```
@Interned String query;
```

A non-null List of English Strings

```
@NonNull List<@English String> msgs;
```

A non-empty array of English strings

```
@English String @NotEmpty [] a;
```



Java 8 extends annotation syntax

Annotations on all occurrences of types:

```
@Untainted String query;  
List<@NonNull String> strings;  
myGraph = (@Immutable Graph) tmp;  
class UnmodifiableList<T>  
    implements @ReadOnly List<T> {}
```

Stored in classfile

Handled by javac, javap, javadoc, ...



Java 6 & 7 compatibility

Annotations in comments:

```
List</*@Nonnull*/ String> strings;
```

(Requires use of jsr308-langtools compiler.)



Array annotations

A **read-only array** of **non-empty arrays** of **English strings**:

```
@English String @ReadOnly [] @NonEmpty [] a;
```



Explicit method receivers

```
class MyClass {  
    int foo(@TParam String p) {...}  
    int foo(@TRecv MyClass this,  
            @TParam String p) {...}
```

No impact on method binding and overloading



Constructor return & receiver types

Every constructor has a return type

```
class MyClass {  
    @TReturn MyClass(@TParam String p) {...}
```

Inner class constructors also have a receiver

```
class Outer {  
    class Inner {  
        @TReturn Inner(@TRecv Outer Outer.this,  
                        @TParam String p) {...}
```



Annotating external libraries

When type-checking clients, need library spec

Can write manually or automatically infer

Two syntaxes:

- As separate text file (stub file)
- Within its .jar file (from annotated partial source code)



Checker Framework facilities

- Full type systems: inheritance, overriding, ...
- Generics (type polymorphism)
 - Also qualifier polymorphism
- Qualifier defaults
- Dataflow framework
- Pre-/post-conditions
- Warning suppression
- Testing infrastructure



Dataflow Framework

Initially for flow-sensitive type refinement

Now a project of its own right

1. Translate AST to CFG

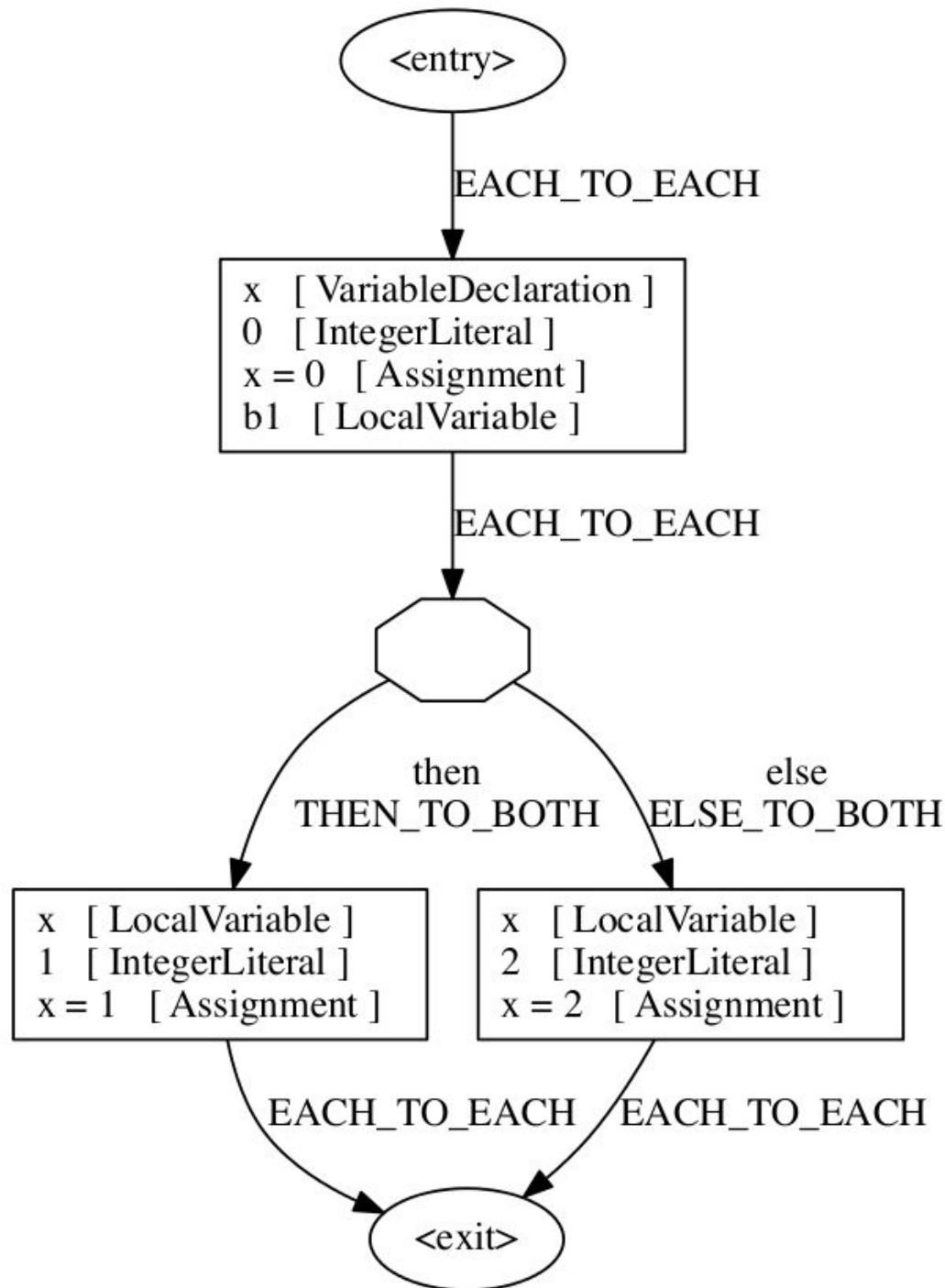
Standard multipass visitor over AST

2. Perform dataflow analysis over CFG with user-provided

- a. Abstract value What are we tracking?
- b. Transfer functions What do operations do?
- c. Store What are intermediate results?

3. Allow queries about result





Properties of the CFG

- Explicit representation of implicit Java constructs
 - Unboxing, implicit type conversions, etc.
 - Analyses do not need to worry about these things
 - All control flow explicitly modeled (e.g. exceptions on field access)
- High-level constructs
 - Close to source language
- Different from other approaches
 - Not three-address-form
 - Analysis is not performed over the AST

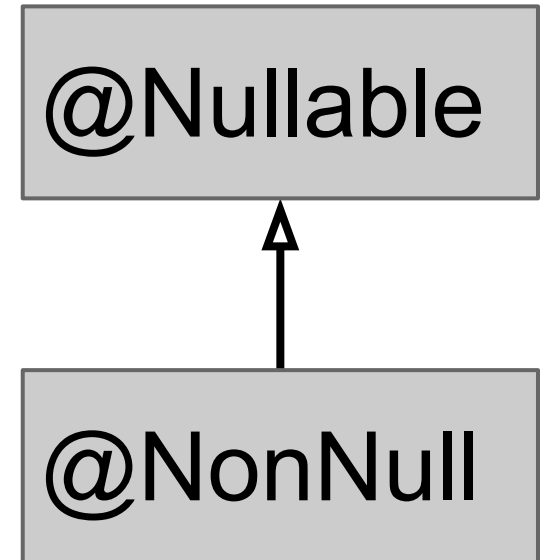


Preventing Null-Pointer Exceptions

Basic type system:

`@Nullable` might be null

`@NonNull` non-null



Default is `@NonNull`

(Opposite of Java's assumption)

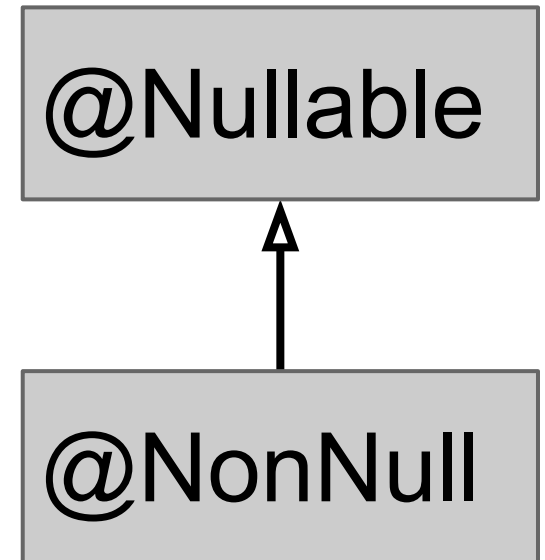
- Makes the dangerous case explicit
- Requires fewer annotations



CLIMB-to-top defaulting rule (applies to all type systems)

Top type is the default for:

- **C**asts
- **L**ocal variables
- **I**ntanceof
- **iM**plicit **B**ounds



Type refinement from assignments

```
myVar = new Foo();
```

Likewise for cast/instanceof expression

Top for implicit types allows every instantiation



Dynamic checks

```
if (x != null) {  
    x.hashCode();  
}
```

```
if (!RegexUtil.isRegex(userInput)) {  
    throw new RuntimeException(...);  
}
```

```
Pattern p = Pattern.compile(userInput);
```



One check for `null` is not enough

```
if (myField != null) {  
    method1();  
    myField.hashCode();  
}
```

3 ways to express persistence across side effects:

```
@SideEffectFree void method1() { ... }
```

```
@EnsuresNonNull("myField") method1()  
{...}
```

```
@MonotonicNonNull myField;
```



Side effects

`@SideEffectFree`

Does not modify externally-visible state

`@Deterministic`

If called with `== args` again, gives `== result`

`@Pure`

Both side-effect-free and deterministic

The side-effect annotations are trusted, not checked



Lazy initialization and persistence across side effects

`@MonotonicNonNull`

Might be null or non-null

May only be (re-)assigned a non-null value

Purpose: avoid re-checking

Once non-null, always non-null



Method pre- and post-conditions

Preconditions:

`@RequiresNonNull`

Postconditions:

`@EnsuresNonNull`

`@EnsuresNonNullIf`

```
@EnsuresNonNullIf(expression="#1", result=true)  
public boolean equals(@Nullable Object obj) { ... }
```



Polymorphism over qualifiers

@PolyNull

Each occurrence is a use of an implicitly-defined type qualifier variable

@PolyAll

Same for all type systems



A non-null field might contain null

```
@NonNull String name;  
... myObject.name ...
```

Initialization

`@Initialized` (constructor has completed)

`@UnderInitialization(Frame.class)`

Its constructor is currently executing

`@UnknownInitialization`

Might be initialized or under initialization



Map keys and Map.get

```
Map<String, @NonNull Integer> gifts;  
... gifts.get("pipers piping") ...
```

Map.get can return null unless

- value type is non-null, **and**
- argument key appears in the map

@KeyFor [rarely written, usually inferred]



Suppressing warnings

Because of Checker Framework false positives

`@SuppressWarnings("nullness")`

Use smallest possible scope (e.g., local var)

Write the rationale

```
assert x != null : "@AssumeAssertion(nullness)";
```

More: <http://types.cs.washington.edu/checker-framework/current/checker-framework-manual.html#suppressing-warnings>



Optional checks

- `Alint=redundantNullComparison`

Warns if comparing a non-null value to null

- `Alint=uninitialized`

Warns if the constructor does not initialize all fields (even primitives that have a default)



Brainstorming new type checkers

What runtime exceptions to prevent?

What properties of data should always hold?

What operations are legal and illegal?

Type-system checkable properties:

- Dependency on values
- Not on program structure, timing, ...



Example: Nullness Checker

What runtime exceptions to prevent?

What properties of data should always hold?

What operations are legal and illegal?



Example: Nullness Checker

What runtime exceptions to prevent?

NullPointerException

What properties of data should always hold?

What operations are legal and illegal?



Example: Nullness Checker

What runtime exceptions to prevent?

NullPointerException

What properties of data should always hold?

@NonNull references always non-null

What operations are legal and illegal?



Example: Nullness Checker

What runtime exceptions to prevent?

NullPointerException

What properties of data should always hold?

@NonNull references always non-null

What operations are legal and illegal?

Dereferences only on @NonNull references



Example: Regex Checker

What runtime exceptions to prevent?

What properties of data should always hold?

What operations are legal and illegal?



Example: Regex Checker

What runtime exceptions to prevent?

PatternSyntaxException,
IndexOutOfBoundsException

What properties of data should always hold?

What operations are legal and illegal?



Example: Regex Checker

What runtime exceptions to prevent?

PatternSyntaxException,
IndexOutOfBoundsException

What properties of data should always hold?

Whether a string is a regex and number of groups

What operations are legal and illegal?



Example: Regex Checker

What runtime exceptions to prevent?

PatternSyntaxException,
IndexOutOfBoundsException

What properties of data should always hold?

Whether a string is a regex and number of groups

What operations are legal and illegal?

Pattern.compile with non-@Regexp, etc,



New type system

What runtime exceptions to prevent?

1

What properties of data should always hold?

2

What operations are legal and illegal?

3



New type system

What runtime exceptions to prevent?

1

What properties of data should always hold?

2

What operations are legal and illegal?

3



New type system

What runtime exceptions to prevent?

1

What properties of data should always hold?

2

What operations are legal and illegal?

3



Building a checker is easy

Example: Ensure encrypted communication

```
void send(@Encrypted String msg) {...}  
@Encrypted String msg1 = ...;  
send(msg1);    // OK  
String msg2 = .....;  
send(msg2);    // Warning!
```



Building a checker is easy

Example: Ensure encrypted communication

```
void send(@Encrypted String msg) {...}  
@Encrypted String msg1 = ...;  
send(msg1);    // OK  
String msg2 = .....;  
send(msg2);    // Warning!
```

The complete checker:

```
@Target(ElementType.TYPE_USE)  
@SubtypeOf(Unqualified.class)  
public @interface Encrypted {}
```



Defining a type system

1. Qualifier hierarchy
 - defines subtyping
2. Type introduction rules
 - types for expressions, declarations
3. Type rules
 - checker-specific errors
4. Flow-refinement
 - better types than the programmer wrote



Defining a type system

1. Qualifier hierarchy
 - subtyping, assignments

```
@SubtypeOf(UnknownRegex.class)  
public @interface Regex {
```



Defining a type system

2. Type introduction rules

- types for expressions, declarations

```
@ImplicitFor( trees = {  
    Tree.Kind.NEW_CLASS,  
    Tree.Kind.NEW_ARRAY, ... })
```

```
@DefaultQualifierInHierarchy
```

```
@DefaultForUnannotatedCode({  
    DL.PARAMETERS, DL.LOWER_BOUNDS })
```



Defining a type system

3. Type rules

- checker-specific errors

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



Defining a type system

4. Flow-refinement

- better types than the programmer wrote

```
if (ElementUtils.matchesElement(method,  
    IS_REGEX_METHOD_NAME,  
    String.class, int.class)) {  
    ...  
}
```



Testing infrastructure

jtreg-based testing as in OpenJDK

Lightweight tests with in-line expected errors:

```
String s = "%+s%";  
//:: error: (format.string.invalid)  
f.format(s, "illegal");
```



Tips

- Start by type-checking part of your code
- Only type-check properties that matter to you
- Use subclasses (not type qualifiers) if possible
- Write the spec first (and think of it as a spec)
- Avoid warning suppressions when possible
- Avoid raw types such as `List`; use `List<String>`



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.



Community

Open source project:

<https://github.com/typetools/checker-framework>

Community:

- uWashington: Michael Ernst, Suzanne Millstein, Javier Thaine, Dan Brown ...
- uWaterloo: Werner Dietl, Jeff Luo, Jason Li, Mier Ta, Charles Chen, ...
- Bug reports, test cases, patches, ... from users



Conclusions

Checker Framework for creating type checkers

- Featureful, effective, easy to use, scalable

Prevent bugs at compile time

Create custom type-checkers

Improve your code!



<http://CheckerFramework.org/>

@CheckerFrmwrk on Twitter
CheckerFramework on Facebook & Google+

