

Preventing Errors Before They Happen

The Checker Framework



<http://CheckerFramework.org/>

Twitter: @CheckerFrmwrk

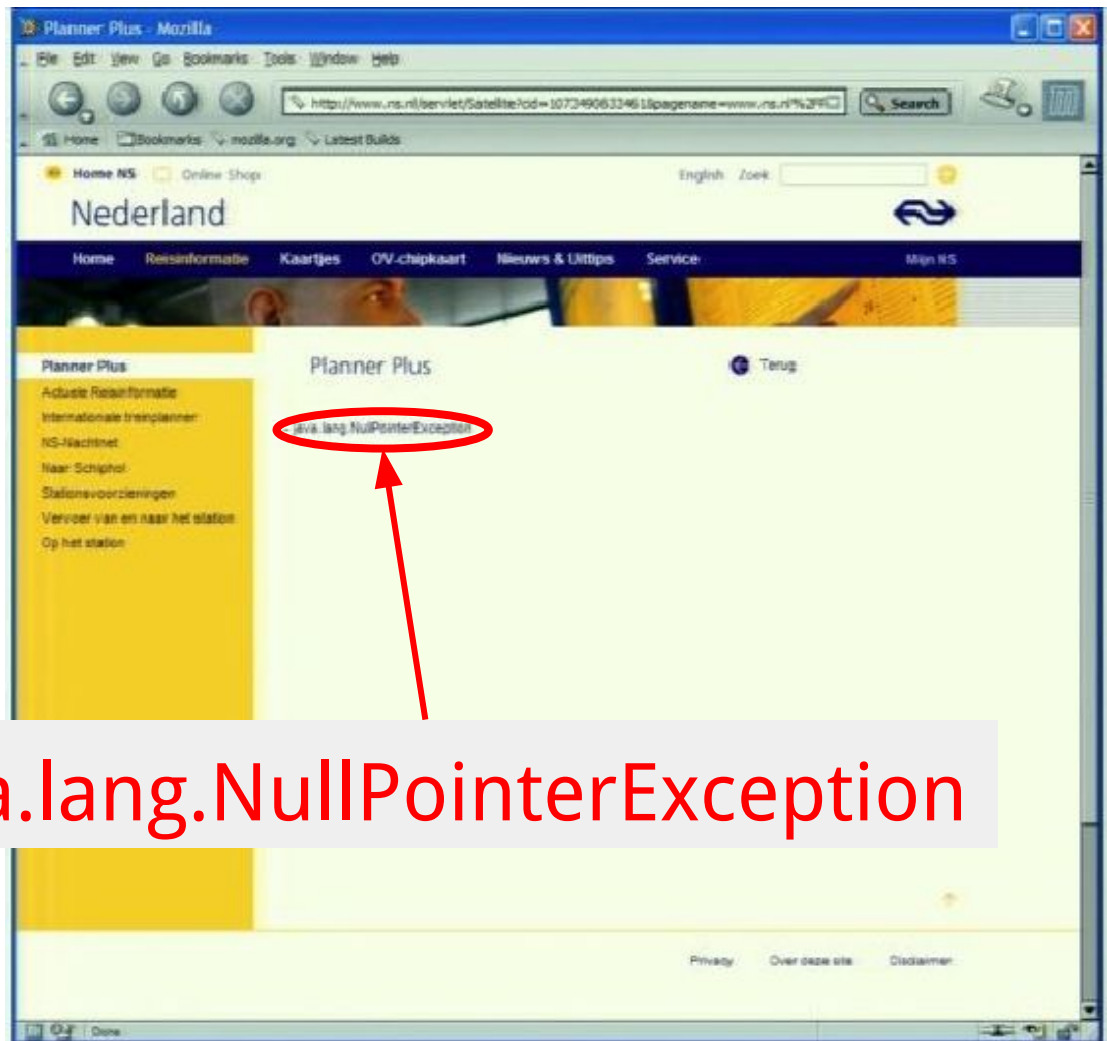
Facebook/Google+: CheckerFramework

Werner Dietl, University of Waterloo

Michael Ernst, University of Washington



Motivation



`java.lang.NullPointerException`



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

1997: 225 deaths: jet crash caused by radar software

1991: 28 deaths: Patriot missile guidance system

2003: 11 deaths: blackout

1985-2000: >8 deaths: Radiation therapy

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



Outline

- Solution: Pluggable type-checking
- Tool: Checker Framework
- How to use it
- Creating your own type system



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

```
NullPointerException
```

```
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
```

```
UnsupportedOperationException
```

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



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);
```

Corrupted map



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)

```
@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;  
                ^
```

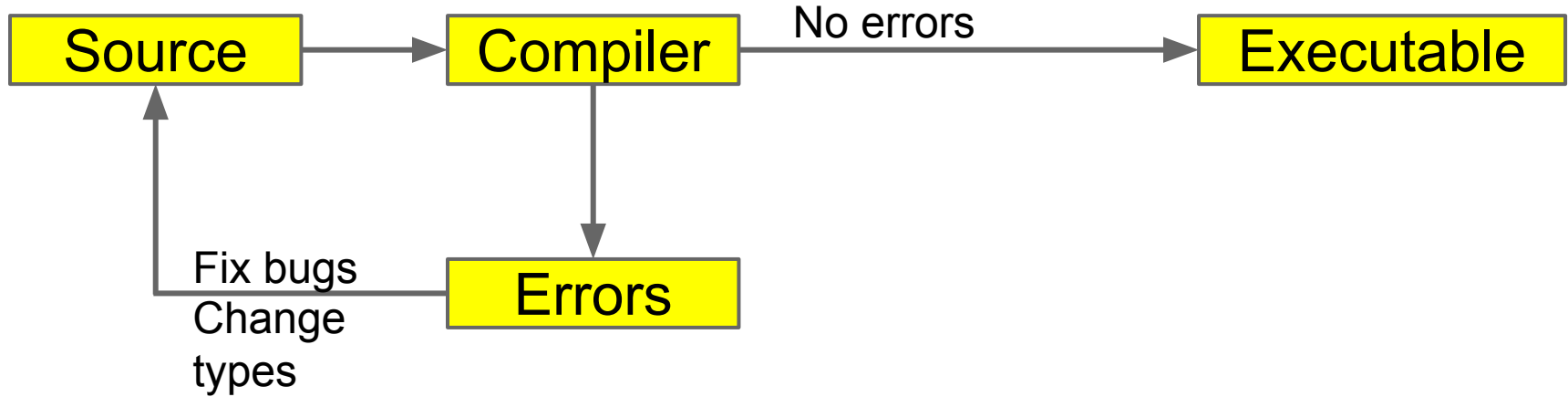


Nullness and mutation demo

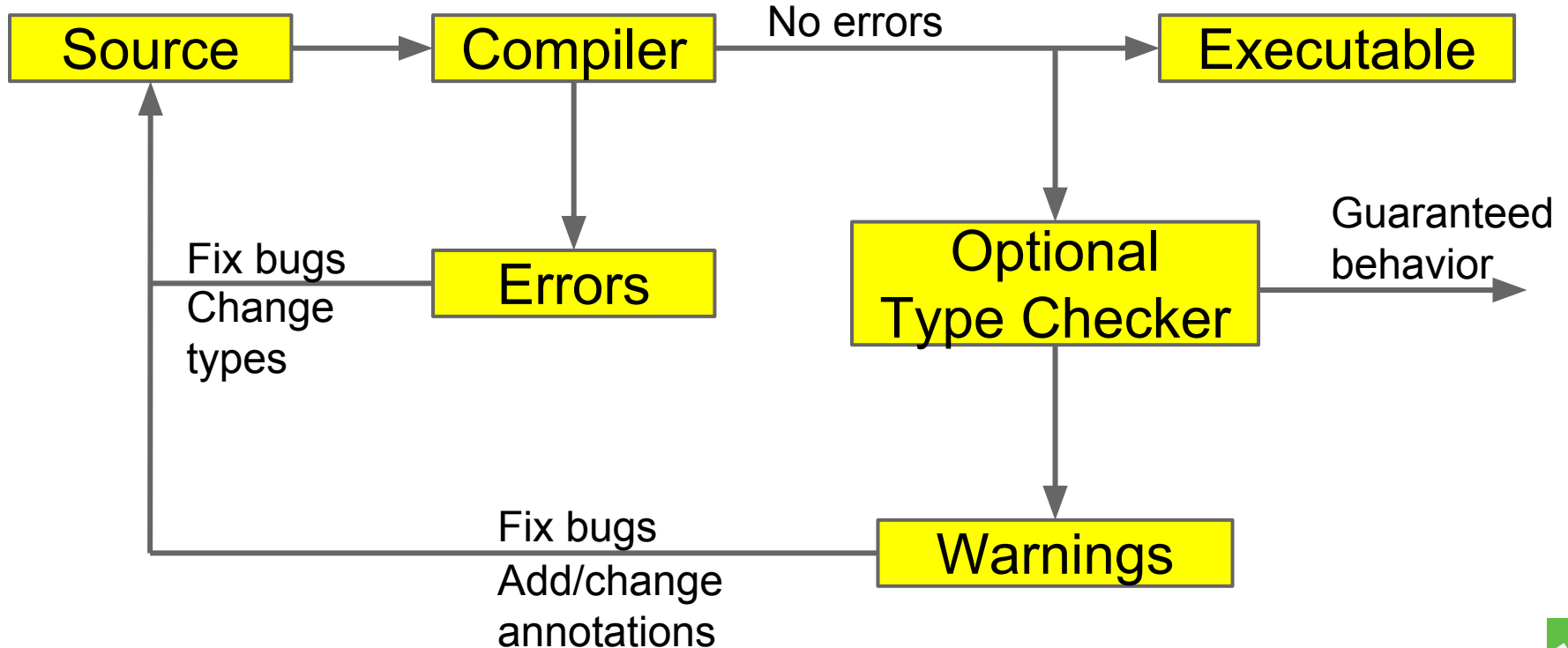
- Detect errors
- Guarantee the absence of errors
- Verify the correctness of optimizations



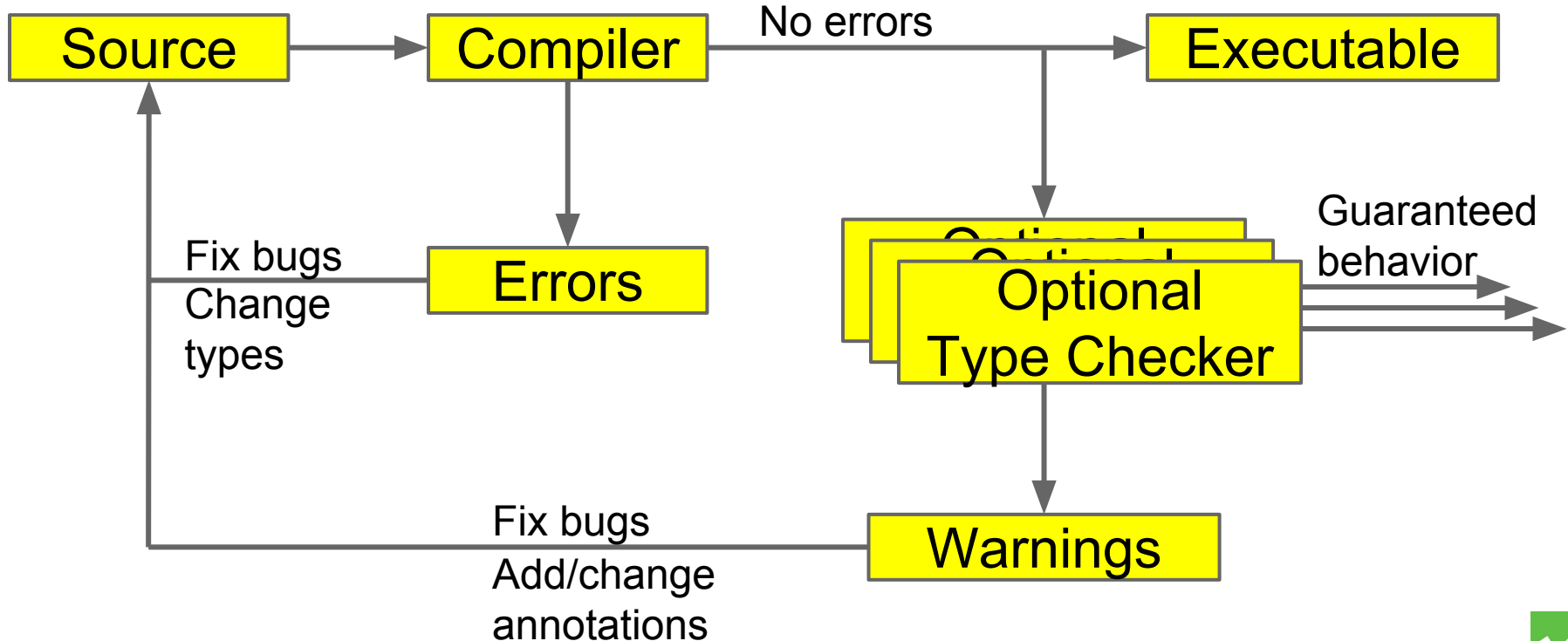
Type Checking



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);           // error
```



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
- 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  
    String conten  
    Pattern pat = Pattern.compile(regex);  
    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

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>
```



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, ...

Command injection vulnerabilities (`@OsTrusted`)

5 missing validations in Hadoop

Privacy (`@Source`)

SPARTA detected malware in Android apps

Learn more at 12:30 today



More example 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

Fake enumerations (`@Fenum`)

problems in Swing, JabRef

You can write your own checker!



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 (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

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



Practicality

Testing

Built-in Type
Systems

Pluggable
Type Systems

Formal
Verification

Guarantees



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.)



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)



Dataflow Framework

Goal: Compute properties about expressions

- More accurate types than the user wrote
- Constant value analysis
- Many other uses, e.g. by Google error-prone

User provides:

- Abstract value What are we tracking?
- Transfer functions What do operations do?
- Store What are intermediate results?

Dataflow framework does all the work!

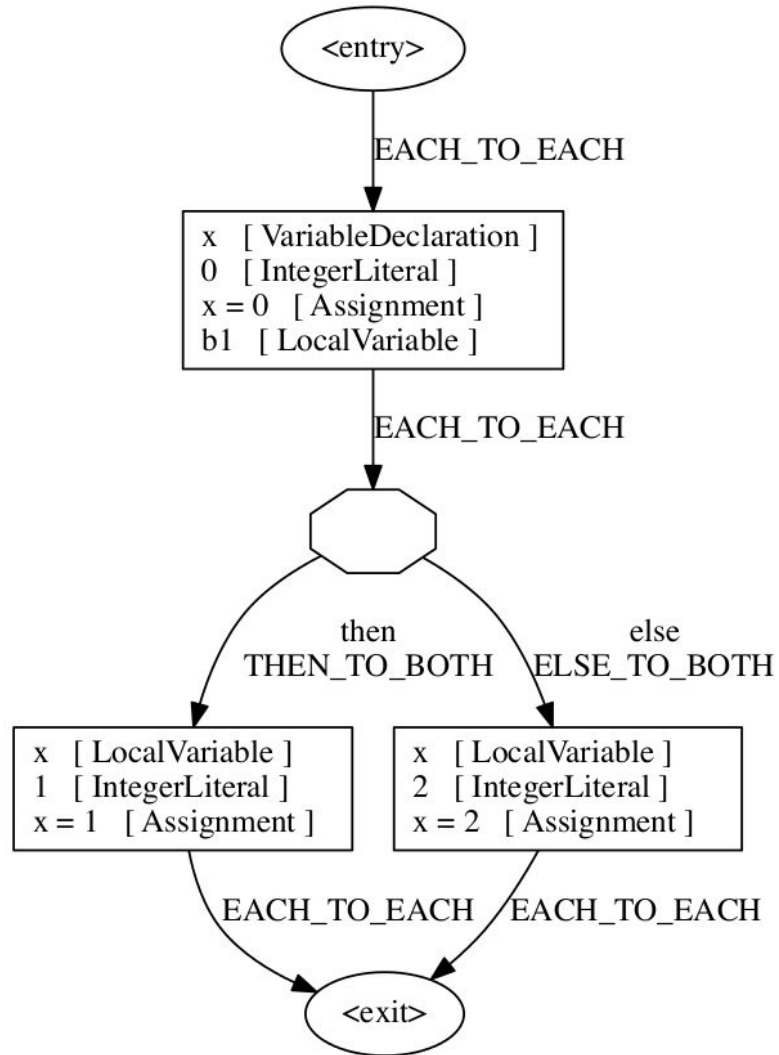


Dataflow Framework

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)





Checker Framework facilities

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



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 `IndexOutOfBoundsException`

What properties of data should always hold?

2 lower bound on length
upper bound on index

What operations are legal and illegal?

3 `index < length && index >= 0`



New type system

What runtime exceptions to prevent?

1 Injection attacks

What properties of data should always hold?

2 Trusted vs. Untrusted data

What operations are legal and illegal?

3 Trusted ops only with trusted data



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 {}
```



Encrypted Checker Demo

Let's build it!



Defining a type system

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



Defining a type system

1. Qualifier hierarchy

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



Defining a type system

2. Type introduction rules

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

```
@DefaultQualifierInHierarchy
```

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



Defining a type system

3. Type rules

```
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

```
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.



More at JavaOne 2015

Collaborative Verification of the Information
Flow for a High-Assurance App Store

CON 4444, today, 12:30 to 13:30

Using Type Annotations to Improve Your Code

BoF 4390, tonight, 20:00 to 20:45



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/>

