Preventing Errors Before They Happen

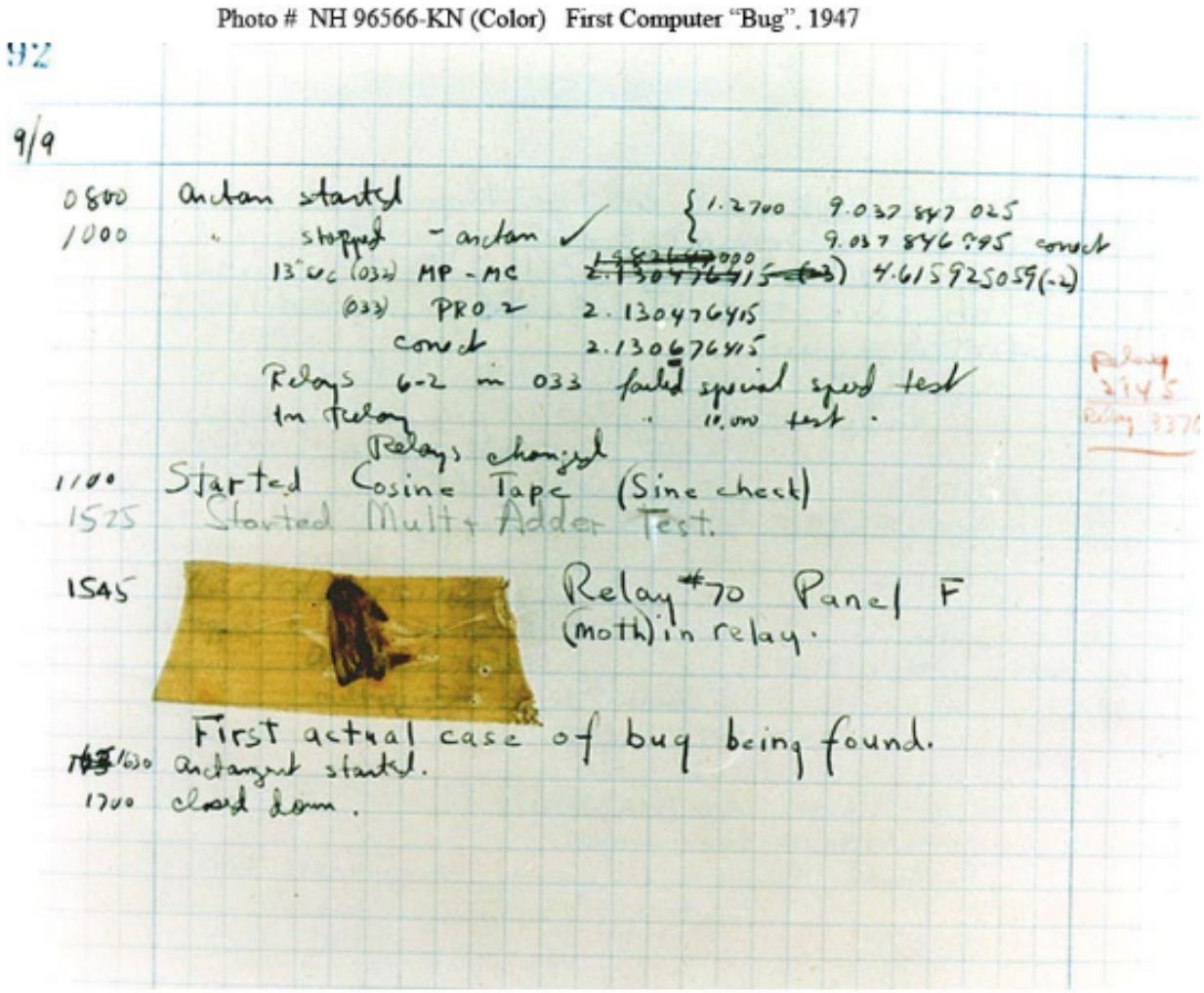
http://CheckerFramework.org/

Werner Dietl
University of Waterloo
https://ece.uwaterloo.ca/~wdietl/

UNIVERSITY OF WATERLOO

Joint work with Michael D. Ernst and many others.

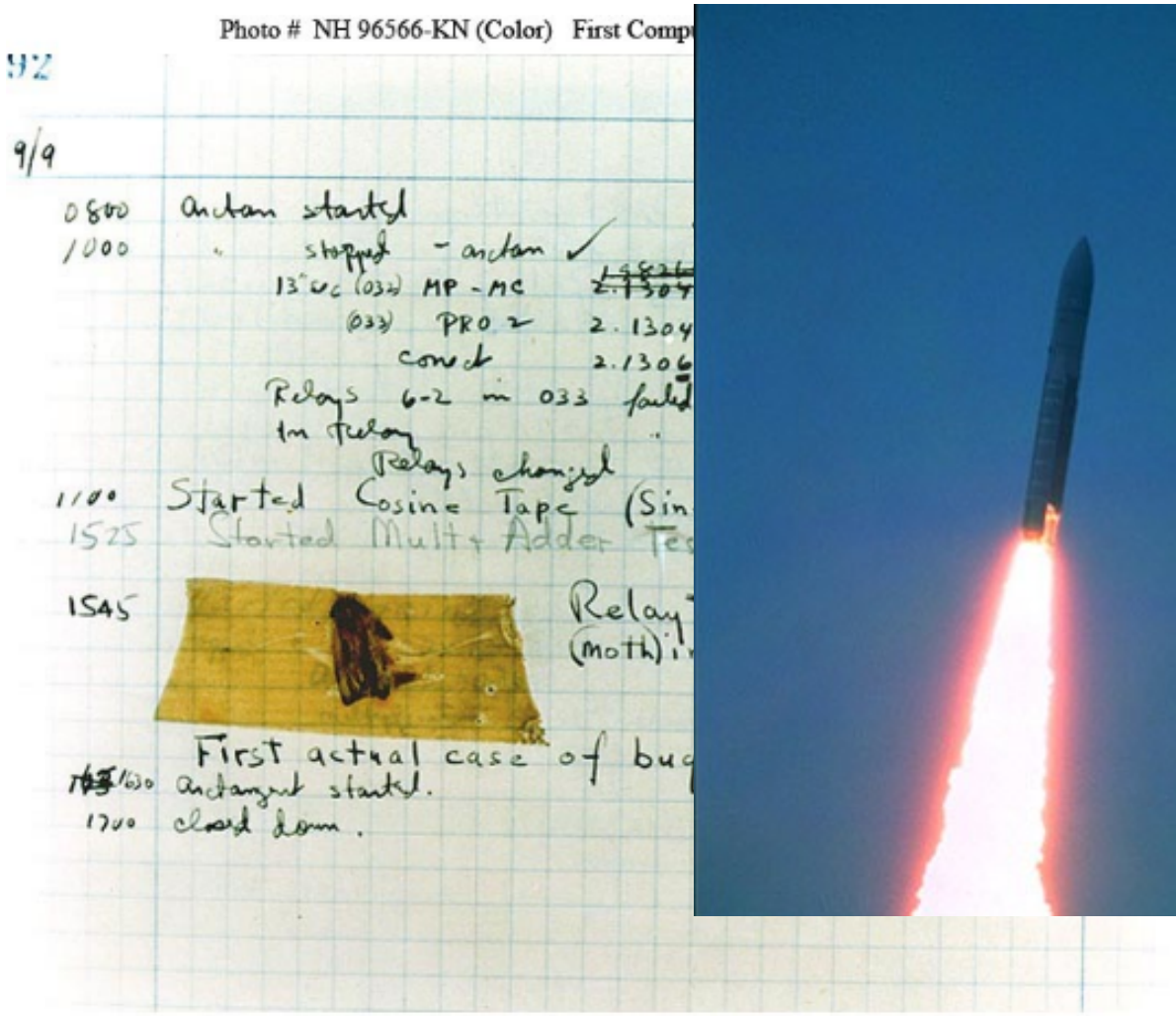
Bug Evolution



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

UNIVERSITY OF WATERLOO

Bug Evolution



http://dlimages.businessweek.com/imageserve/0arRbTIdRSg3e/630x418.jpg

Cost of software failures

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

$300 billion dealing with the Y2K problem

$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

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

Software bugs can cost lives

225 deaths in 1997: jet crash caused by radar software

28 deaths in 1991: Patriot missile guiding system failure

Radiation therapy

- at least three deaths in 1985-87 by Therac-25
- at least five deaths in 2000 by Multidata

11 deaths and $6 billion estimated damages in 2003: blackout in Northeast USA & Canada; 50 million people without power; monitoring software bug

Software cause for quarter of all medical device recalls in 2011

November 2012 medical pump recall

Outline

- Pluggable type-checking
- Quick demo
- Architecture overview
  - Java 8 syntax for Type Annotations
  - Checker Framework
- Writing your own type system
- Nullness Checker in detail
- Our experience
  - SPARTA: Android Security
  - Verigames: Crowd-sourced Verification Games

Outline

- Pluggable type-checking
- Quick demo
- Architecture overview
  - Java 8 syntax for Type Annotations
  - Checker Framework
- Writing your own type system
- Nullness Checker in detail
- Our experience
  - SPARTA: Android Security
  - Verigames: Crowd-sourced Verification Games

Java's type system is too weak

Type checking prevents many errors

```
int i = “hello”; // error
```

Type checking doesn't prevent enough errors

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

Collections.emptyList().add(“one”);

dbStatement.executeQuery(userData);
```

Java's type system is too weak

Type checking prevents many errors

```
int i = “hello”; // error
```

Type checking c ors

NullPointerException

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

Collections.emptyList().add(“one”);

dbStatement.executeQuery(userData);
```

Java's type system is too weak

Type checking prevents many errors

```
int i = “hello”; // error
```

Type checking doesn't prevent enough errors

```
System
Collections.emptyList().add(“one”);
dbStatement.executeQuery(userData);
```

UnsupportedOperationException

Java's type system is too weak

Type checking prevents many errors

```
int i = “hello”; // error
```

Type checking doesn't prevent enough errors

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

Collections

dbStatement.executeQuery(userData);
```

SQL Injection Attacks!

Null pointer exception

```
String op(Data in) {
    return “transform: ” + in.getF();
}
...
String s = op(null);
```

Where is the error?

Null pointer exception

```
String op(Data in) {
    return “transform: ” + in.getF();
}
...
String s = op(null);
```

Where is the error?

Can’t decide without specification!

Solution 1: Restrict use

```
String op(@NonNull Data in) {
    return “transform: ” + in.getF();
}
…
String s = op(null);      // error
```

Solution 2: Restrict implementation

```
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
  - Reduce number of run-time checks

Possible negatives:

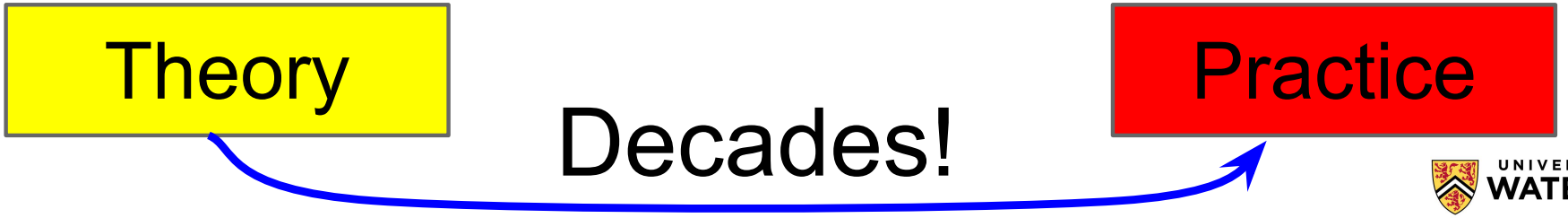
- Must write the types (or use type inference)
- False positives are possible (can be suppressed)

Extended type systems

- Null pointer exceptions (ICSE SEIP'11)
- Energy consumption (PLDI'11)
- Unwanted side effects (OOPSLA'04, '05, '12, ECOOP'08, ESEC/FSE’07)
- Unstructured heaps (ECOOP'07, '11, '12)
- Malformed input (FTfJP'12)
- UI actions not on event thread (ECOOP'13)
- Information leakage (CCS’14)

Extended type systems

- Null pointer exceptions (ICSE SEIP'11)
- Energy consumption (PLDI'11)
- Unwanted side effects (OOPSLA'04, '05, '12, ECOOP'08, ESEC/FSE'07)
- Unstructured heaps (ECOOP'07, '11, '12)
- Malformed input (FTfJP'12)
- UI actions not on event thread (ECOOP'13)
- Information leakage (CCS'14)



The Checker Framework

A framework for pluggable type checkers
"Plugs" into the OpenJDK compiler

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

Eclipse plug-in, Ant and Maven integration

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 content
  Pattern pat = Pattern.compile(regex);
  Matcher mat = pat.matcher(content);

  if (ma
    System.out.p      in( "Group: "
          mat.group(1));
  }
}
```

PatternSyntaxException

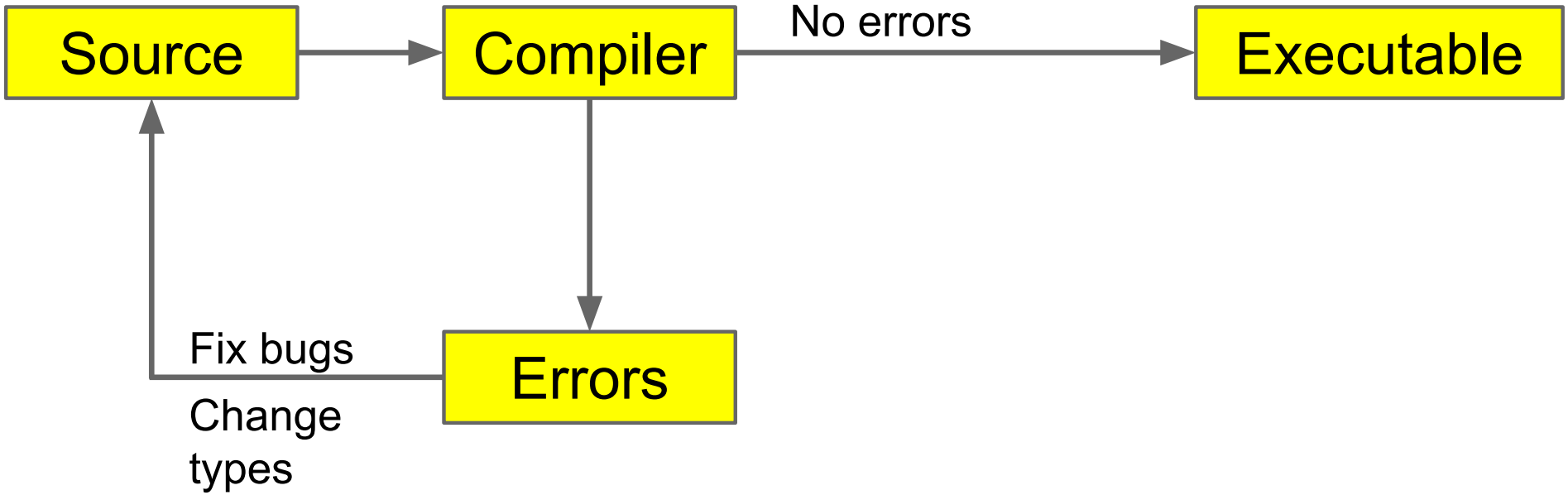
IndexOutOfBoundsExceptionon

Fixing the Errors

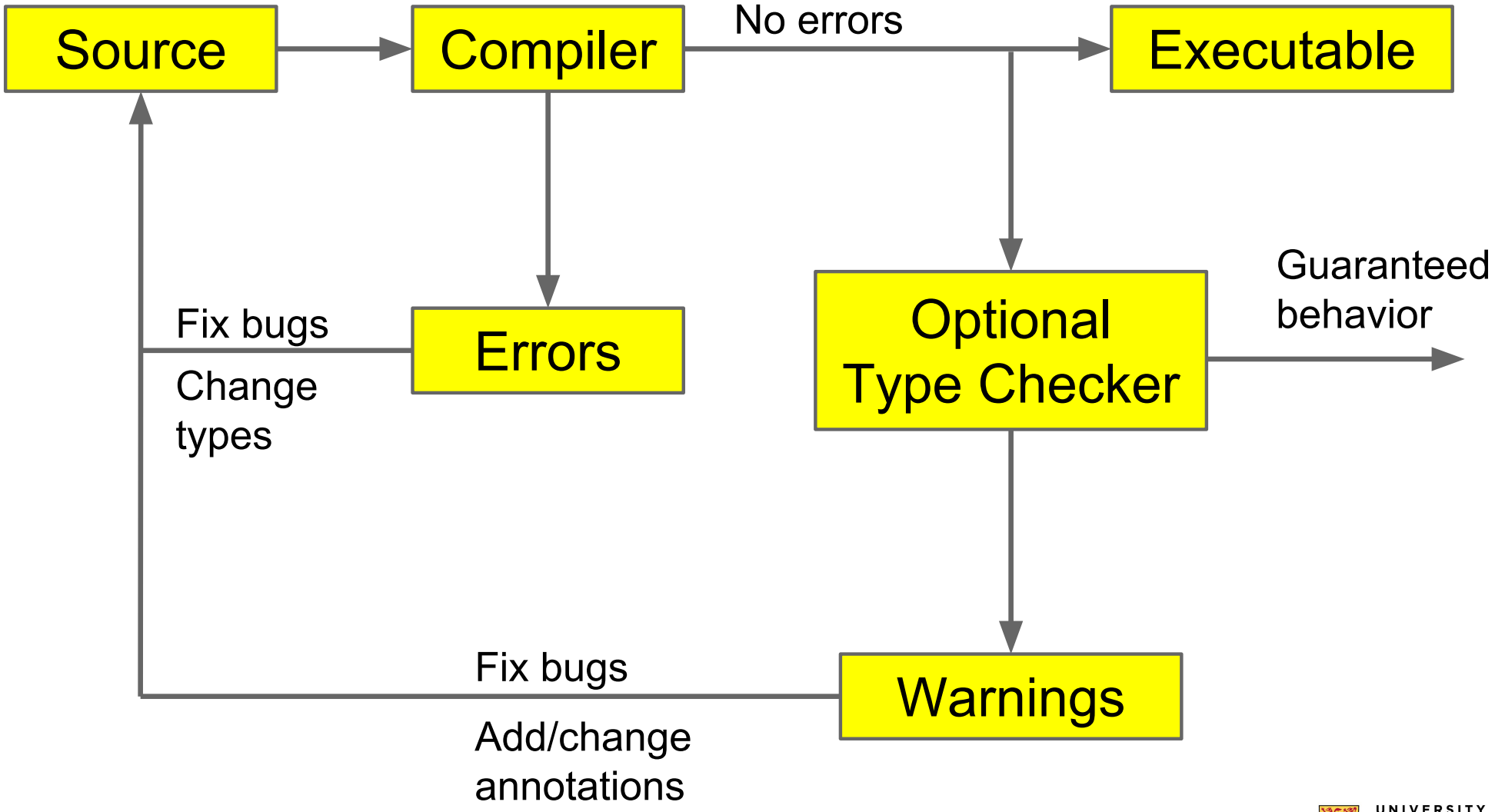
`Pattern.compile` only on valid regex
`Matcher.group(i)` only if > i groups

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

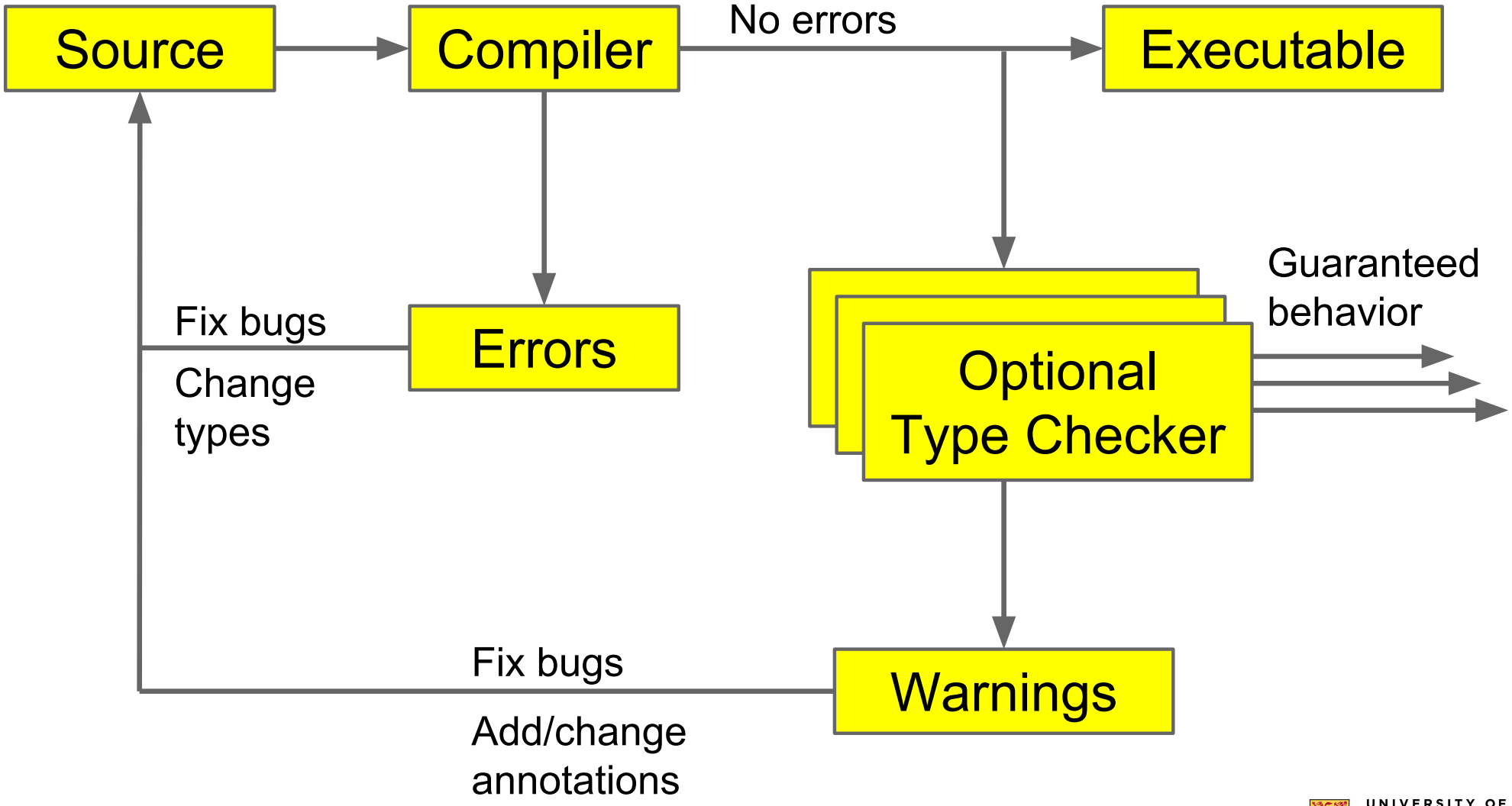
Type Checking



Optional Type Checking



Optional Type Checking



What bugs can you detect & prevent?

The property you care about:	The annotation you write:
Null dereferences	`@NonNull`
Mutation and side-effects	`@Immutable`
Concurrency: locking	`@GuardedBy`
Security: encryption,	`@Encrypted`
tainting	`@Untainted`
Aliasing	`@Linear`
Equality tests	`@Interned`
Strings: localization,	`@Localized`
Regular expression syntax	`@Regex`
Signature format	`@FullyQualified`
Enumerations	`@Fenum`
Typestate (e.g., open/closed files)	`@State`

Users can write their own checkers!

UNIVERSITY OF WATERLOO

Checker Framework experience

Type checkers reveal important latent defects

Ran on >6 million LOC of real-world open-source code

Found hundreds of user-visible failures

Improved design and documentation

Annotation overhead is low

Mean 2.6 annotations per kLOC [Dietl et al. ICSE'11]

Formalizations

$$
\begin{array}{rcll}
\mathtt{h} & \in & \mathtt{Heap} & = \mathtt{Addr} \rightarrow \mathtt{Obj} \\
\iota & \in & \mathtt{Addr} & = \text{Set of Addresses} \cup \{\mathtt{null}_a\} \\
\mathtt{o} & \in & \mathtt{Obj} & = {}^{\mathtt{r}}\mathtt{Type},\ \mathtt{Fields} \\
{}^{\mathtt{r}}\mathtt{T} & \in & {}^{\mathtt{r}}\mathtt{Type} & = \mathtt{OwnerAddr}\ \mathtt{ClassId}\langle\overline{{}^{\mathtt{r}}\mathtt{Type}}\rangle \\
\mathtt{Fs} & \in & \mathtt{Fields} & = \mathtt{FieldId} \rightarrow \mathtt{Addr} \\
\iota & \in & \mathtt{OwnerAddr} & = \mathtt{Addr} \cup \{\mathtt{any}_a\} \\
{}^{\mathtt{r}}\Gamma & \in & {}^{\mathtt{r}}\mathtt{Env} & = \overline{\mathtt{TVarId}\ {}^{\mathtt{r}}\mathtt{Type}};\ \overline{\mathtt{ParId}\ \mathtt{Addr}}
\end{array}
$$

$$
\begin{array}{rcll}
\mathtt{P} & \in & \mathtt{Program} & ::= \overline{\mathtt{Class}},\ \mathtt{ClassId},\ \mathtt{Expr} \\
\mathtt{Cls} & \in & \mathtt{Class} & ::= \mathtt{class}\ \mathtt{ClassId}\langle\overline{\mathtt{TVarI}}\rangle \\
 & & & \quad \mathtt{extends}\ \mathtt{ClassId}\langle\overline{{}^{\mathtt{s}}\mathtt{Ty}}\rangle \\
 & & & \quad \{\ \overline{\mathtt{FieldId}\ {}^{\mathtt{s}}\mathtt{Type}};\ \overline{\mathtt{Met}} \\
{}^{\mathtt{s}}\mathtt{T} & \in & {}^{\mathtt{s}}\mathtt{Type} & ::= {}^{\mathtt{s}}\mathtt{NType} \mid \mathtt{TVarId} \\
{}^{\mathtt{s}}\mathtt{N} & \in & {}^{\mathtt{s}}\mathtt{NType} & ::= \mathtt{OM}\ \mathtt{ClassId}\langle\overline{{}^{\mathtt{s}}\mathtt{Type}}\rangle \\
\mathtt{u} & \in & \mathtt{OM} & ::= \\
\mathtt{mt} & \in & \mathtt{Meth} & ::= \\
 & & \mathtt{MethSig} & ::= \\
\mathtt{w} & \in & \mathtt{Purity} & ::= \\
\mathtt{e} & \in & \mathtt{Expr} & ::= \\
 & & & \quad \mathtt{Expr.MethId}\langle{}^{\mathtt{s}}\mathtt{Type}\rangle(\mathtt{Expr}) \mid \\
 & & & \quad \mathtt{new}\ {}^{\mathtt{s}}\mathtt{Type} \mid (\,{}^{\mathtt{s}}\mathtt{Type})\ \mathtt{Expr} \\
{}^{\mathtt{s}}\Gamma & \in & {}^{\mathtt{s}}\mathtt{Env} & ::= \overline{\mathtt{TVarId}\ {}^{\mathtt{s}}\mathtt{NType}};\ \overline{\mathtt{ParId}\ {}^{\mathtt{s}}\mathtt{Type}}
\end{array}
$$

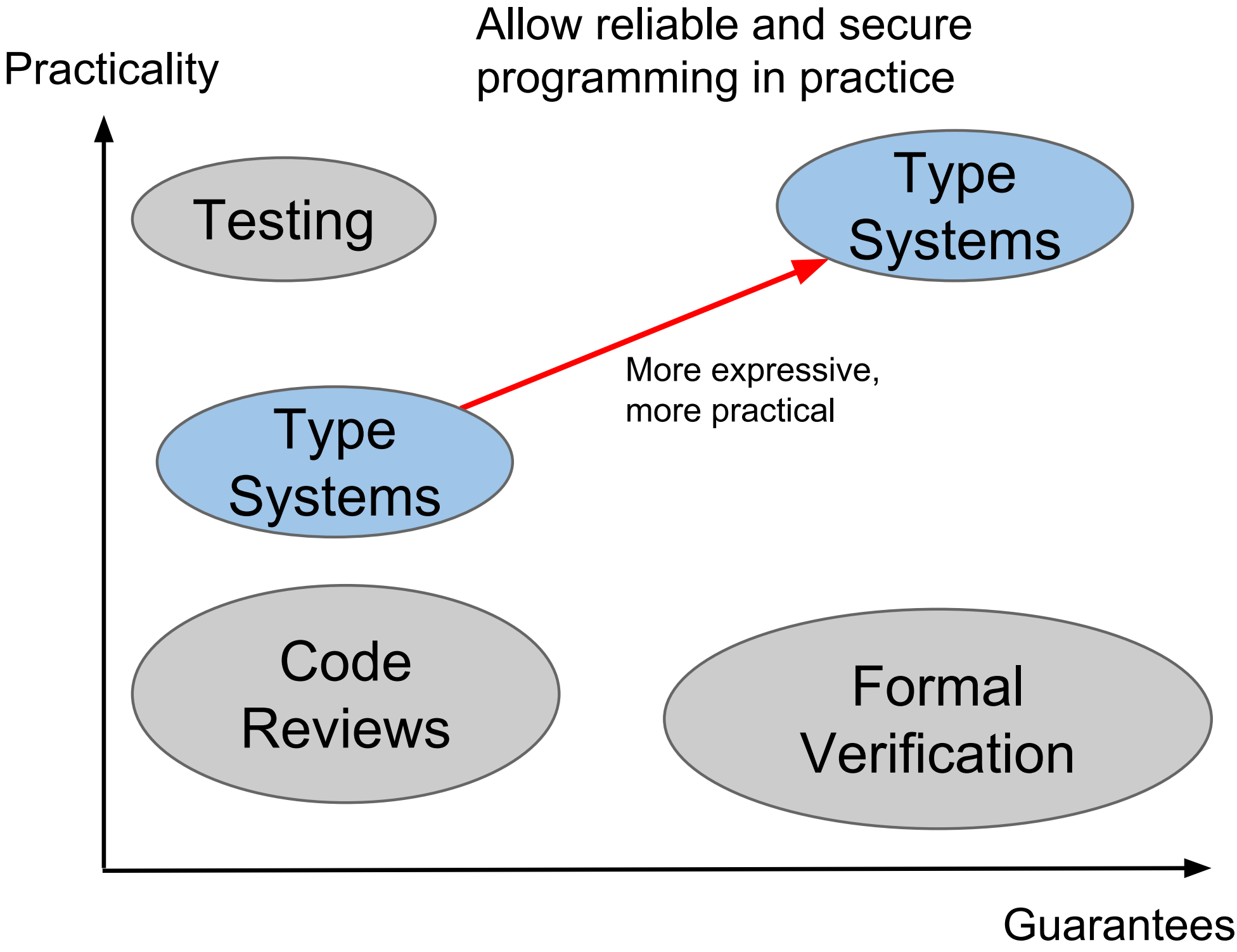
$$\text{OS-Upd}\ \frac{\begin{array}{c}\mathtt{h}, {}^{\mathtt{r}}\Gamma, \mathtt{e}_0 \rightsquigarrow \mathtt{h}_0, \iota_0 \\ \iota_0 \neq \mathtt{null}_a \\ \mathtt{h}_0, {}^{\mathtt{r}}\Gamma, \mathtt{e}_2 \rightsquigarrow \mathtt{h}_2, \iota \\ \mathtt{h}' = \mathtt{h}_2[\iota_0.\mathtt{f} := \iota]\end{array}}{\mathtt{h}, {}^{\mathtt{r}}\Gamma, \mathtt{e}_0.\mathtt{f} = \mathtt{e}_2 \rightsquigarrow \mathtt{h}',}$$

$$\text{OS-Read}\ \frac{\begin{array}{c}\mathtt{h}, {}^{\mathtt{r}}\Gamma, \mathtt{e}_0 \rightsquigarrow \mathtt{h}', \iota_0 \\ \iota_0 \neq \mathtt{null}_a \\ \iota = \mathtt{h}'(\iota_0)\downarrow_2(\mathtt{f})\end{array}}{\mathtt{h}, {}^{\mathtt{r}}\Gamma, \mathtt{e}_0.\mathtt{f} \rightsquigarrow \mathtt{h}', \iota}$$

$$\text{GT-Read}\ \frac{\Gamma \vdash \mathtt{e}_0 : \mathtt{N}_0 \qquad \mathtt{N}_0 = \_\ (}{\Gamma \vdash \mathtt{e}_0.\mathtt{f} : \mathtt{N}_0 \rhd \mathit{fType}(\mathtt{C}_0, \mathtt{f})}$$

$$\text{GT-Upd}\ \frac{\begin{array}{c}\Gamma \vdash \mathtt{e}_0 : \mathtt{N}_0 \qquad \mathtt{N}_0 = \mathtt{u}_0\ \mathtt{C}_0\langle\_\rangle \\ \mathtt{T}_1 = \mathit{fType}(\mathtt{C}_0, \mathtt{f}) \\ \Gamma \vdash \mathtt{e}_2 : \mathtt{N}_0 \rhd \mathtt{T}_1 \\ \mathtt{u}_0 \neq \mathtt{any} \qquad \mathit{rp}(\mathtt{u}_0, \mathtt{T}_1)\end{array}}{\Gamma \vdash \mathtt{e}_0.\mathtt{f} = \mathtt{e}_2 : \mathtt{N}_0 \rhd \mathtt{T}_1}$$

$$\text{DYN}\ \frac{\begin{array}{l}\mathtt{h} \vdash {}^{\mathtt{r}}\Gamma : {}^{\mathtt{s}}\Gamma \\ \mathtt{h} \vdash \iota_1 : \mathit{dyn}({}^{\mathtt{s}}\mathtt{N}, \mathtt{h}, {}^{\mathtt{r}}\Gamma) \\ \mathtt{h} \vdash \iota_2 : \mathit{dyn}({}^{\mathtt{s}}\mathtt{T}, \iota_1, \mathtt{h}(\iota_1)\downarrow_1) \\ {}^{\mathtt{s}}\mathtt{N} = \mathtt{u}_N\ \mathtt{C}_N\langle\_\rangle \\ \mathtt{u}_N = \mathtt{this}_u \Rightarrow {}^{\mathtt{r}}\Gamma(\mathtt{this}) \\ \mathit{free}({}^{\mathtt{s}}\mathtt{T}) \subseteq \mathit{dom}(\mathtt{C}_N)\end{array} \Longrightarrow \mathtt{h} \vdash \iota_2 : \mathit{dyn}({}^{\mathtt{s}}\mathtt{N} \rhd {}^{\mathtt{s}}\mathtt{T}, \mathtt{h}, {}^{\mathtt{r}}\Gamma) \qquad \begin{array}{c}{}^{\mathtt{r}}\mathtt{T} = \iota'\ \_\langle\_\rangle \qquad \iota \vdash {}^{\mathtt{r}}\mathtt{T}\ {}^{\mathtt{r}}{<:}\ \iota'\ \mathtt{C}\langle\overline{{}^{\mathtt{r}}\mathtt{T}}\rangle \qquad \iota \vdash {}^{\mathtt{r}}\mathtt{T}\ {}^{\mathtt{r}}{<:}\ \iota'\ \mathtt{C}\langle\overline{{}^{\mathtt{r}}\mathtt{T}_a}\rangle \Rightarrow \iota \vdash \overline{{}^{\mathtt{r}}\mathtt{T}}\ {}^{\mathtt{r}}{<:}\ \overline{{}^{\mathtt{r}}\mathtt{T}_a} \\ \mathrm{dom}(\mathtt{C}) = \overline{\mathtt{X}} \qquad \mathrm{free}({}^{\mathtt{s}}\mathtt{T}) \subseteq \overline{\mathtt{X}} \circ \overline{\mathtt{X}'}\end{array}}{\mathit{dyn}({}^{\mathtt{s}}\mathtt{T}, \iota, {}^{\mathtt{r}}\mathtt{T}, (\overline{\mathtt{X}'\ {}^{\mathtt{r}}\mathtt{T}'}; \_)) = {}^{\mathtt{s}}\mathtt{T}[\iota'/\mathtt{this}, \iota'/\mathtt{peer}, \iota/\mathtt{rep}, \mathtt{any}_a/\mathtt{any}_u, \overline{{}^{\mathtt{r}}\mathtt{T}/\mathtt{X}}, \overline{{}^{\mathtt{r}}\mathtt{T}'/\mathtt{X}'}]}$$


Allow reliable and secure programming in practice
Practicality
Testing
Type Systems
More expressive, more practical
Type Systems
Code Reviews
Formal Verification
Guarantees

Outline

- Pluggable type-checking
- Quick demo
- Architecture overview
  - Java 8 syntax for Type Annotations
  - Checker Framework
- Writing your own type system
- Nullness Checker in detail
- Our experience
  - SPARTA: Android Security
  - Verigames: Crowd-sourced Verification Games

Since Java 5: 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 read-only array of non-empty arrays 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 read-only array of non-empty arrays of English strings:

```
@English String @ReadOnly [] @NonEmpty [] 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, …

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) {...}
```

CF: Java 6 & 7 Compatibility

Annotations in comments

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

Voodoo comments for arbitrary source code

```
/*>>> import myquals.TRecv; */
...
  int foo(/*>>> @TRecv MyClass this,*/
          @TParam String p) {…}
```

CF: stub files and annotated JDK

Allows annotating external libraries

- As separate text file (stub file)

  checker-framework/checker/src/…/jdk.astub

- Within its .jar file (from annotated partial source code)

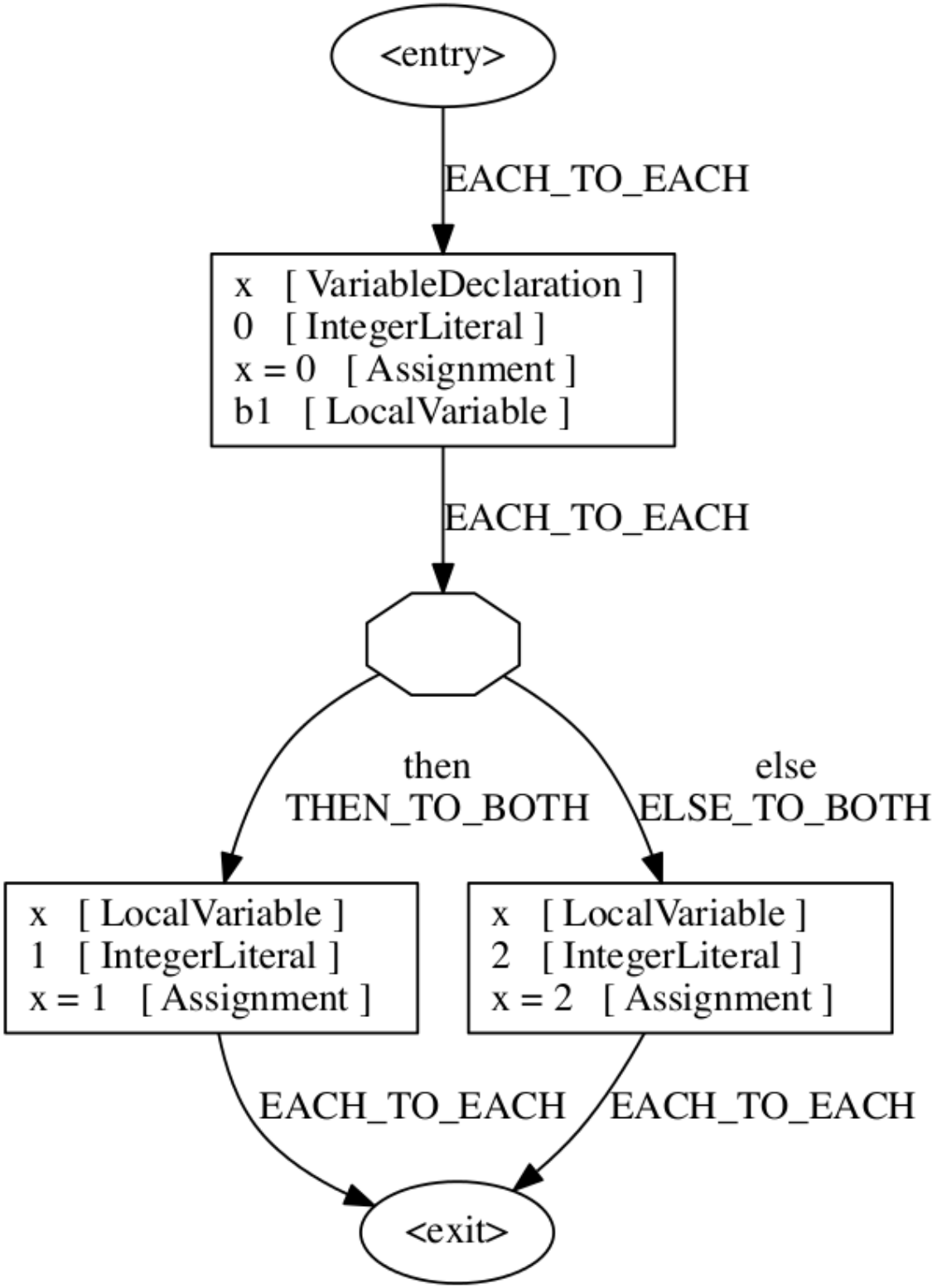
  checker-framework/checker/jdk/...

Dataflow Framework

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



Checker Framework - “Framework”

- Full type systems: inheritance, overriding, ...
- Generics (type polymorphism)
  - Also qualifier polymorphism
- Flow-sensitive type qualifier inference
  - Infers types for local variables
  - reusable component
- Qualifier defaults
- Pre-/Post-conditions
- Warning suppression
- Testing infrastructure

Checker Framework type systems

The property you care about:	The annotation you write:
Null dereferences	`@NonNull`
Mutation and side-effects	`@Immutable`
Concurrency: locking	`@GuardedBy`
Security: encryption,	`@Encrypted`
tainting	`@Untainted`
Aliasing	`@Linear`
Equality tests	`@Interned`
Strings: localization,	`@Localized`
Regular expression syntax	`@Regex`
Signature format	`@FullyQualified`
Enumerations	`@Fenum`
Typestate (e.g., open/closed files)	`@State`

Outline

- Pluggable type-checking
- Quick demo
- Architecture overview
  - Java 8 syntax for Type Annotations
  - Checker Framework
- Writing your own type system
- Nullness Checker in detail
- Our experience
  - SPARTA: Android Security
  - Verigames: Crowd-sourced Verification Games

Building checkers is easy

Example: Ensure encrypted communication

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

Building checkers 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 {}
```

Let’s try this!

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
   - checker-specific flow properties

Testing Infrastructure

jtreg-based testing as in OpenJDK

Light-weight tests with in-line expected errors:

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

Brainstorming new type checkers

What runtime exceptions do you wish 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, ...

Brainstorming

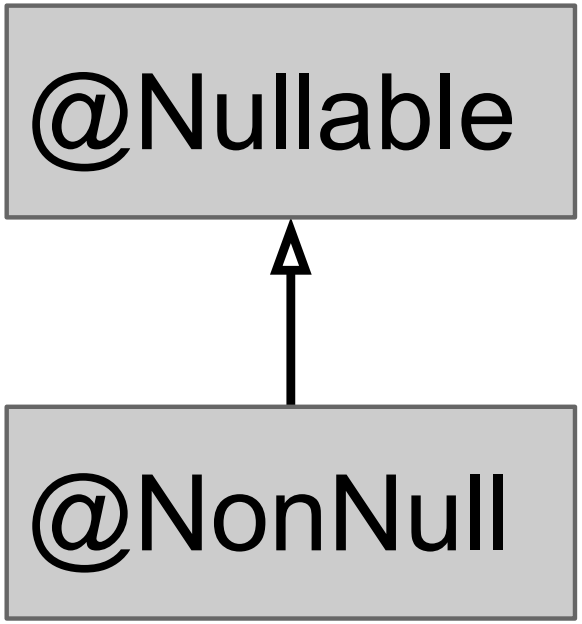
Outline

- Pluggable type-checking
- Quick demo
- Architecture overview
  - Java 8 syntax for Type Annotations
  - Checker Framework
- Writing your own type system
- Nullness Checker in detail
- Our experience
  - SPARTA: Android Security
  - Verigames: Crowd-sourced Verification Games

Preventing Null-Pointer Exceptions

Basic type system:

`@Nullable`	might be null
`@NonNull`	non-null



Default is `@NonNull`

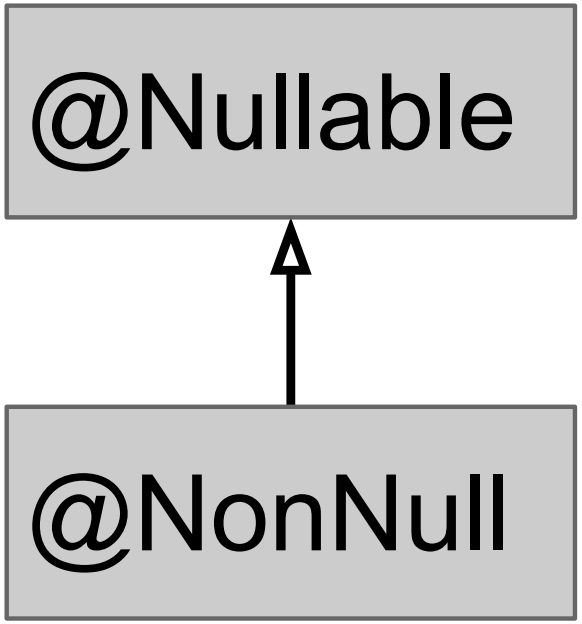
(Opposite of Java's assumption)

- 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**nstanceof
- i**M**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();
}
```

One check for `null` is not enough

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

3 ways to express persistence across side effects:

```
@SideEffectFree void method1() { … }
@MonotonicNonNull myField;
@EnsuresNonNull(“myField”) method1(){…}
```

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) { ... }
```

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

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

How to get started

1. **Write the specification**
   - Search the Javadoc for occurrences of “null”
   - Replace the wordy English text by `@Nullable`
     Can also search code, but no annos in methods
2. **Run Nullness Checker**: verify/complete spec
   - For each warning
     - Reason about why the code is safe
     - Express that reasoning as annotations
     - Consider improving the code’s design

Tips

- 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>`
- Start by type-checking part of your code
- Only type-check properties that matter to you
- Use subclasses (not type qualifiers) if possible

Outline

- Pluggable type-checking
- Quick demo
- Architecture overview
  - Java 8 syntax for Type Annotations
  - Checker Framework
- Writing your own type system
- Nullness Checker in detail
- Our experience
  - SPARTA: Android Security
  - Verigames: Crowd-sourced Verification Games

Evaluations

- Checkers reveal important latent bugs
  - Ran on >6 million LOC of real-world code
  - Found hundreds of user-visible bugs & mistakes
- Annotation overhead is low
  - Mean 2.6 annotations per kLOC
- Learning their usage is easy
  - Used successfully by first-year CS majors
- Building checkers is easy
  - New users developed 3 new realistic checkers

SPARTA: Static Program Analysis for Reliable Trusted Apps

Security type system for Android apps
Guarantees no leakage of private information

Part of
Automated Program Analysis
for Cybersecurity (APAC)
program.
See CCS'14 paper

UNIVERSITY OF WATERLOO

Crowd-sourced verification

Make software verification easy and fun

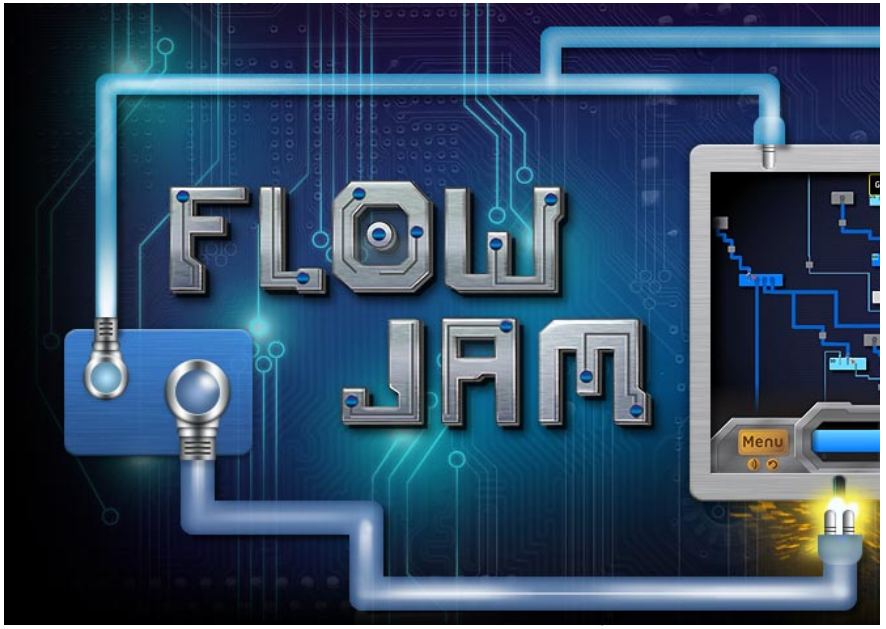
Make the game accessible to everyone

Harness the power of the crowd

Goal: Verify software while waiting

`http://verigames.com/`

FTfJP'12 paper



More at JavaOne'14

Using Type Annotations to Improve Code Quality

BoF, tonight, 19:00 to 19:45

Enhanced Metadata in Java SE 8

Conference talk, Wednesday, 16:30 to 17:30

Conclusions

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 at:

`http://CheckerFramework.org/`