Automated Repair of Pluggable Type Checker Warnings

Manu Sridharan, Univ. of California, Riverside

Collaborators: Nima Karimipour, Sanjay Malakar, Justin Pham, Narges Shadab (UCR), Pascal Joos, Michael Pradel (CISPA), Martin Kellogg (NJIT), Michael Ernst (U. of Washington), Pritam Gharat, Shrey Tiwari, Shuvendu K. Lahiri, Akash Lal (Microsoft Research), Lazaro Clapp (Uber)

UCR
UNIVERSITY OF CALIFORNIA, RIVERSIDE

Pluggable Types

Practical Pluggable Types for Java

Matthew M. Papi Mahmood Ali Telmo Luis Correa Jr. Jeff H. Perkins Michael D. Ernst
MIT Computer Science and Artificial Intelligence Lab, Cambridge, MA, USA
{mpapi,mali,telmo,jhp,mernst}@csail.mit.edu

Abstract

This paper introduces the Checker Framework, which supports adding pluggable type systems to the Java language in a backward-compatible way. A type system designer defines type qualifiers and their semantics, and a compiler plug-in enforces the semantics. Programmers can write the type qualifiers in their programs and use the plug-in to detect or prevent errors. The Checker Framework is useful both to programmers who wish to write error-free code, and to type system designers who wish to evaluate and deploy their type systems.

The Checker Framework includes new Java syntax for expressing type qualifiers; declarative and procedural mechanisms for writing type-checking rules; and support for flow-sensitive local type qualifier inference and for polymorphism over types and qualifiers. The Checker Framework is well-integrated with the Java language and toolset.

We have evaluated the Checker Framework by writing 5 checkers and running them on over 600K lines of existing code. The checkers found real errors, then confirmed the absence of further errors in the fixed code. The case studies also shed light on the type systems themselves.

Categories and subject descriptors: D3.3 [Programming Languages]: Language Constructs and Features—*data types and structures*; F3.1 [Logics and Meanings of Programs]: Specifying and Verifying and Reasoning about Programs; D1.5 [Programming Techniques]: Object-oriented Programming
General terms: Languages, Theory
Keywords: annotation, bug finding, case study, compiler, flow-sensitivity, IGJ, immutable, intern, Java, javac, Javari, NonNull, pluggable type, polymorphism, readonly, type qualifier, type system, verification

permit more expressive compile-time checking and guarantee the absence of additional errors.

A pluggable type framework serves two key constituencies. It enables a *programmer* to write type qualifiers in a program and to run a type checker that verifies that the program respects the type system. It enables a *type system designer* to define type qualifiers, to specify their semantics, and to create the checker used by the programmer.

Programmers wish to improve the quality of their code without disrupting their workflow. To be useful to programmers, a pluggable type framework should be integrated with the programming language, run-time system, and toolchain. The qualifiers should be a part of the programming language that is supported by every tool and is preserved in compiled executables to the same extent that ordinary types are. An unmodified compiler should be capable of checking the pluggable type system.

Type system designers wish to evaluate and deploy their type systems: a type system is valuable only if it helps programmers, so using a type system in practice is an essential way to gain insight. To be useful to type system designers, a pluggable type framework should make it easy to define simple type systems and possible to define powerful type systems. The framework should declaratively support common type system idioms such as the type hierarchy, implicit type qualifiers, flow-sensitivity, and parametric polymorphism over both types and qualifiers. However, the framework should also support procedural specification of features specific to the type system. Furthermore, the declarative and procedural mechanisms should be tightly coupled in a single language. When expressiveness and conciseness are in conflict, expressiveness is more important. A well-known type system is easy enough to understand and implement, whatever the framework, but novel type systems will inevitably require new functionality, and the framework should not be tuned to the type systems of yesterday.

ISSTA 2008



CHECKER framework

checkerframework.org

NullAway

```
static void log(Object x) {
  System.out.println(x.toString());
}
static void foo() {
  log(null);
}
```

NullAway

(@NonNull by default)

```
static void log(Object x) {
  System.out.println(x.toString());
}
static void foo() {
  log(null);
}
```

NullAway

(@NonNull by default)

```
static void log(Object x) {
  System.out.println(x.toString());
}
static void foo() {
  log(null); Error: cannot pass null to @NonNull
}              parameter x
```

NullAway

```
static void log(@Nullable Object x) {
  System.out.println(x.toString());
}
static void foo() {
  log(null);
}
```

NullAway

```
static void log(@Nullable Object x) {
  System.out.println(x.toString());
}
static void foo() {
  log(null);
}
```

Error: de-referencing x may yield NPE

NullAway

```
static void log(@Nullable Object x) {
  if (x == null) return;
  System.out.println(x.toString());
}
static void foo() {
  log(null);
}
```

NullAway

```
static void log(@Nullable Object x) {
  if (x == null) return;
  System.out.println(x.toString());
}
static void foo() {
  log(null);
}
```



NullAway

```
static void log(@Nullable Object x) {
  if (x == null) return;
  System.out.println(x.toString());
}
static void foo() {
  log(null);
}
```



Many other examples: resource leaks, tainting, data races, ...

NullAway Adoption

Dec 2025 – Feb 2026



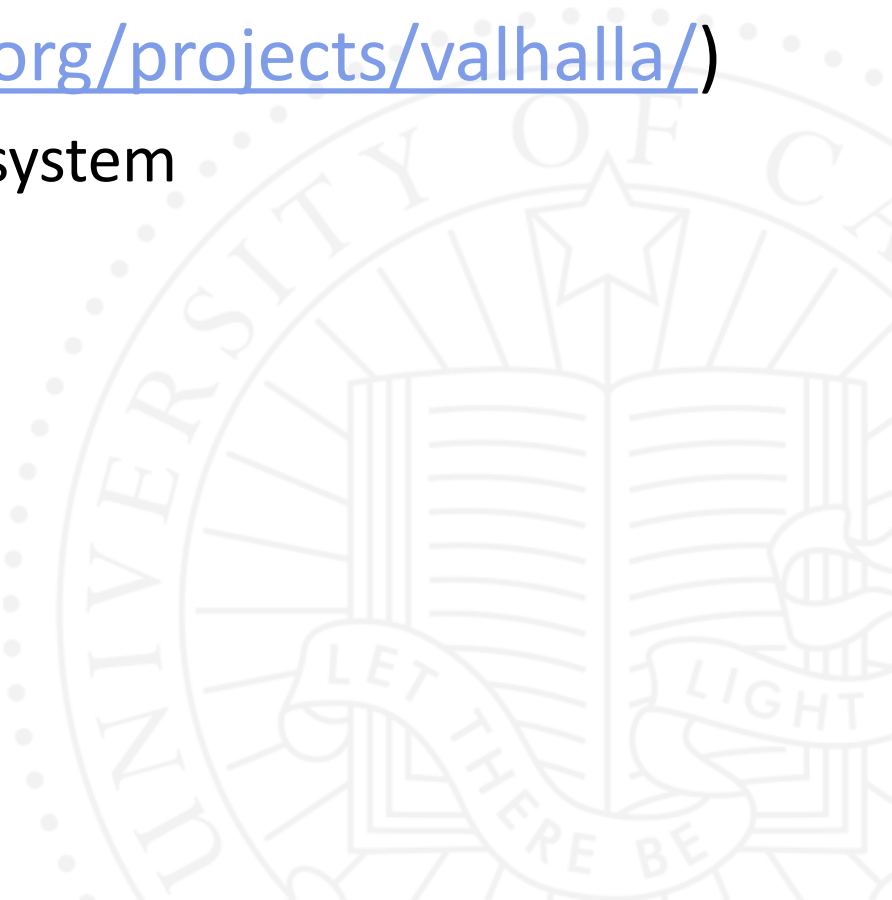
Aug – Sep 2026



- **Recent major projects:** Spring Framework, Micronaut, JUnit, AssertJ, Chrome for Android

Why?

- JSpecify (https://jspecify.dev/)
  - Standardized nullness annotations
- Valhalla (https://openjdk.org/projects/valhalla/)
  - Adds nullness to Java type system
- But also…



Types and Agentic Coding

- Agentic coding works best with strict guardrails
- E.g., thorough test suites
- And…**stronger type systems / static analysis**
- For greenfield projects, NullAway and other pluggable type checkers are a “no brainer”

Challenge: Migrating Existing Code

- We want to **block build on any warning**
  - To stop human / agent mistakes early
- **Problem: need to fix extant warnings!**
  - Add annotations
  - Fix real bugs
  - Refactor or suppress for false positives
- Why not “just use an LLM”?
  - **Costly** for a large code base
  - **Risky** for logic changes

Solution: Automated Repair



Infer annotations to capture intent



Update code to fix bugs, refactor, and suppress



Key techniques:

- **Search**
- **Static analysis**
- **Targeted use of LLMs**

Rest of this talk

- **NullAway Repair**
  - NullAway Annotator (FSE'23): infer annotations
  - NullRepair (ISSTA'26): code changes
  - Ongoing: improved false positive validation
- **Resource Leak Repair**
  - Annotation inference (OOPSLA'23)
  - Arodnap (ASE'25): Improved repair for wrappers
- **Future work**

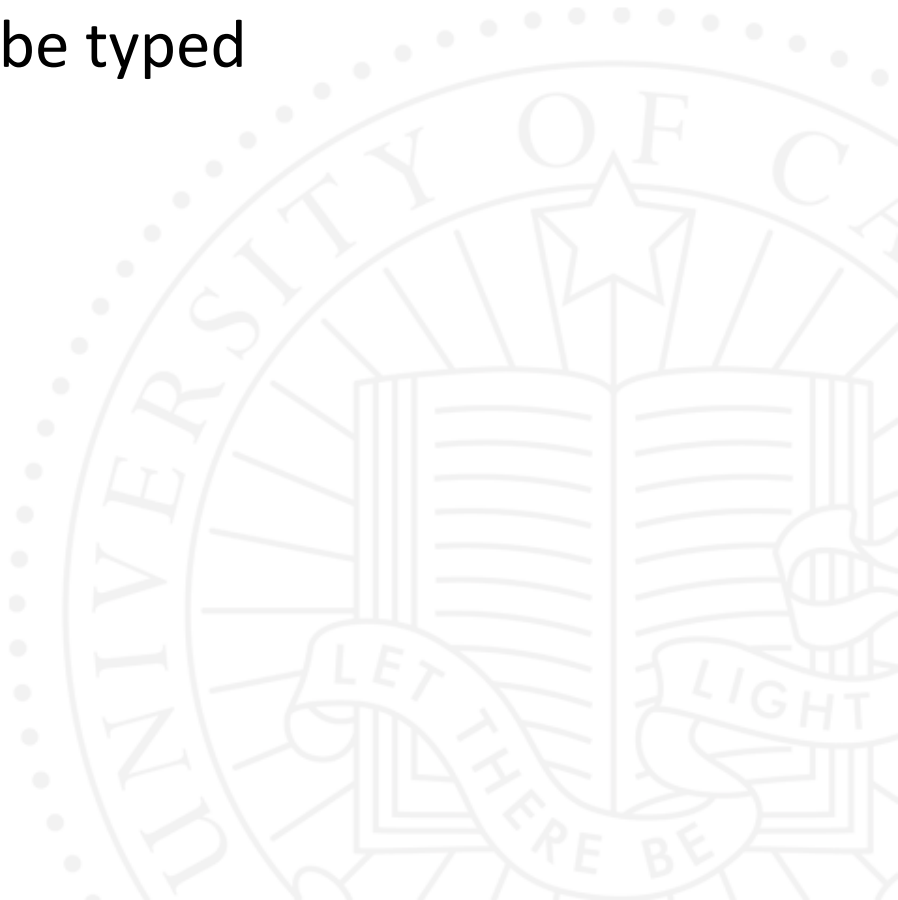
NullAway Annotator (FSE'23)



Requirement: Partial Solutions

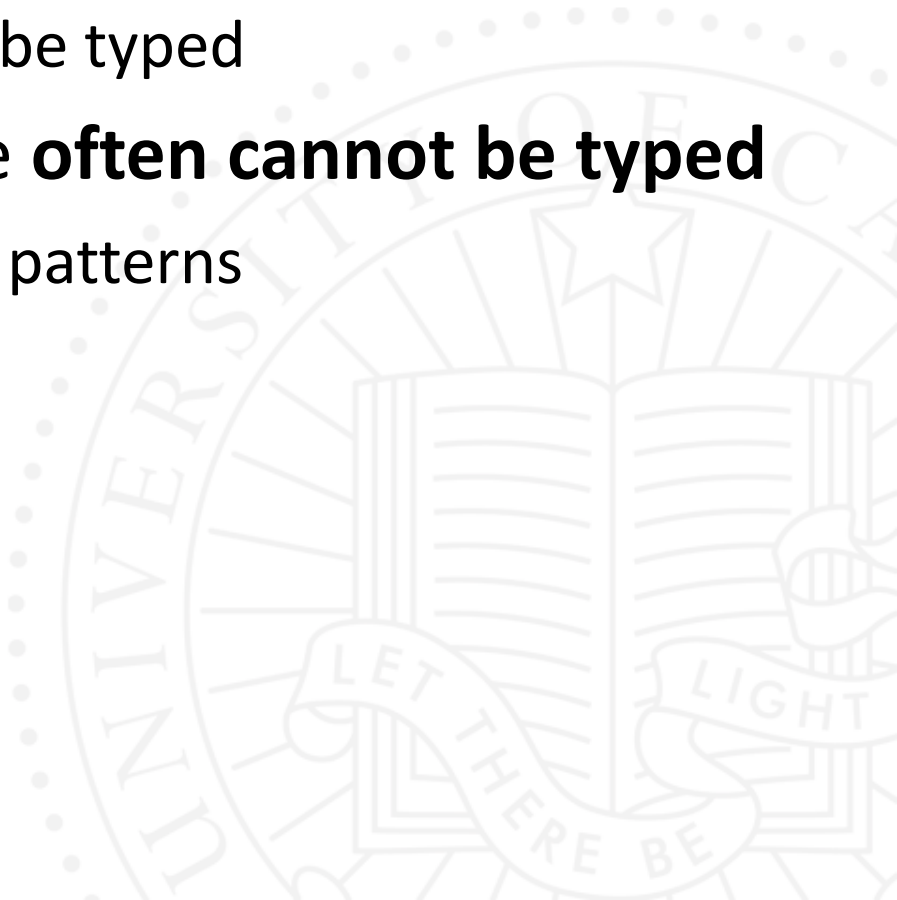
Requirement: Partial Solutions

- Typical type inference outcomes
  - **Success**: valid types for all variables
  - **Failure:** some code cannot be typed



Requirement: Partial Solutions

- Typical type inference outcomes
  - **Success**: valid types for all variables
  - **Failure:** some code cannot be typed
- For nullness, existing code **often cannot be typed**
  - Existing bugs, tricky coding patterns



Requirement: Partial Solutions

- Typical type inference outcomes
  - **Success**: valid types for all variables
  - **Failure:** some code cannot be typed
- For nullness, existing code **often cannot be typed**
  - Existing bugs, tricky coding patterns
- We need **partial solutions**
  - Find types for **as much code as possible**
  - Type checking most (but not all) code has **great value**

More Requirements

- Re-use current NullAway implementation
  - Too risky / costly to directly add inference support
  - **Goal:** treat NullAway as a **black-box oracle**
  - Inference invokes NullAway to test annotations

- Reasonable performance
  - Should run in < 8 hours on a large module

NullAwayAnnotator

- Searches for a set of annotations to **minimize number of remaining NullAway errors**
  - *Maximizes* amount of code that type checks
- For performance, **test multiple annotations simultaneously**
  - Impacts of many annotations *provably independent*
  - Dramatically improves performance
- **Results:** average 69.5% reduction in NullAway errors, good performance (optimizations critical), **over 5 million LoC enrolled** using tool

Example

```
class Test {
            Object f1 = null;
            Object f2 = null;
            Object f3 = null;
            Object f4 = null;
  Object f5 = f4;
  String m1() {
    return f1 != null ? f1.toString() : f2.toString();
  }
  int m2() {
    return f3 != null ? f3.hashCode() : f2.hashCode();
  }
  Object m3() {
    return f4;
  }
}
```

Example

```
class Test {
            Object f1 = null; error
            Object f2 = null; error
            Object f3 = null; error
            Object f4 = null; error
  Object f5 = f4;
  String m1() {
    return f1 != null ? f1.toString() : f2.toString();
  }
  int m2() {
    return f3 != null ? f3.hashCode() : f2.hashCode();
  }
  Object m3() {
    return f4;
  }
}
```

Example

```
class Test {
  +@Nullable Object f1 = null; error
  +@Nullable Object f2 = null; error
  +@Nullable Object f3 = null; error
  +@Nullable Object f4 = null; error
  Object f5 = f4;
  String m1() {
    return f1 != null ? f1.toString() : f2.toString();
  }
  int m2() {
    return f3 != null ? f3.hashCode() : f2.hashCode();
  }
  Object m3() {
    return f4;
  }
}
```

Example

```
class Test {
  +@Nullable Object f1 = null;
  +@Nullable Object f2 = null;
  +@Nullable Object f3 = null;
  +@Nullable Object f4 = null;
  Object f5 = f4;error
  String m1() {
    return f1 != null ? f1.toString() : f2.toString();
  }                                          error
  int m2() {
    return f3 != null ? f3.hashCode() : f2.hashCode();
  }                                          error
  Object m3() {
    return f4; error
  }
}
```

Example

```
class Test {
  +@Nullable Object f1 = null;
  +@Nullable Object f2 = null;
  +@Nullable Object f3 = null;
  +@Nullable Object f4 = null;
  Object f5 = f4;error
  String m1() {
    return f1 != null ? f1.toString() : f2.toString();
  }                                                  error
  int m2() {
    return f3 != null ? f3.hashCode() : f2.hashCode();
  }                                                  error
  Object m3() {
    return f4; error
  }
}
```

Search for error-minimizing solution, invoking NullAway to check candidates

Final Annotations

```
class Test {
  +@Nullable Object f1 = null;
  +@SuppressWarnings("NullAway") Object f2 = null;
  +@Nullable Object f3 = null;
  +@Nullable Object f4 = null;
  +@Nullable Object f5 = f4;
  String m1() {
    return f1 != null ? f1.toString() : f2.toString();
  }
  int m2() {
    return f3 != null ? f3.hashCode() : f2.hashCode();
  }
  +@Nullable Object m3() {
    return f4;
  }
}
```

Final Annotations

```
class Test {
  +@Nullable Object f1 = null;
  +@SuppressWarnings("NullAway") Object f2 = null;
  +@Nullable Object f3 = null;
  +@Nullable Object f4 = null;
  +@Nullable Object f5 = f4;
  String m1() {
    return f1 != null ? f1.toString() : f2.toString();
  }
  int m2() {
    return f3 != null ? f3.hashCode() : f2.hashCode();
  }
  +@Nullable Object m3() {
    return f4;
  }
}
```

Almost all code checkable!

Final Annotations

```
class Test {
  +@Nullable Object f1 = null;
  +@SuppressWarnings("NullAway") Object f2 = null;
  +@Nullable Object f3 = null;
  +@Nullable Object f4 = null;
  +@Nullable Object f5 = f4;
  String m1() {
    return f1 != null ? f1.toString() : f2.toString();
  }
  int m2() {
    return f3 != null ? f3.hashCode() : f2.hashCode();
  }
  +@Nullable Object m3() {
    return f4;
  }
}
```

Almost all code checkable!

But, checking annotations one by one is slow…

Key Optimization: Impacted Regions

- **Insight:** regions where @Nullable annotation can change error count are easy to compute!
  - Due to modular type checking
  - E.g., @Nullable on a field can only change error count in methods accessing the field
- **Idea:** find independent annotations with **graph coloring**
  - Compute conflict graph based on impacted regions
  - Color to determine independence

Optimization Example

```
class Test {
  Object f1 = null;
  Object f2 = null;
  Object f3 = null;
  Object f4 = null;
  Object f5 = f4;
  String m1() {
    return f1 != null ? f1.toString() : f2.toString();
  }
  int m2() {
    return f3 != null ? f3.hashCode() : f2.hashCode();
  }
  Object m3() {
    return f4;
  }
}
```

Optimization Example

```
class Test {
  Object f1 = null;
  Object f2 = null;
  Object f3 = null;
  Object f4 = null;
  Object f5 = f4;
  String m1() {
    return f1 != null ? f1.toString() : f2.toString();
  }
  int m2() {
    return f3 != null ? f3.hashCode() : f2.hashCode();
  }
  Object m3() {
    return f4;
  }
}
```

Impacted Regions

f1 -> m1

f2 -> {m1,m2}

f3 -> m2

f4 -> m3

Optimization Example

```
class Test {
  Object f1 = null;
  Object f2 = null;
  Object f3 = null;
  Object f4 = null;
  Object f5 = f4;
  String m1() {
    return f1 != null ? f1.toString() : f2.toString();
  }
  int m2() {
    return f3 != null ? f3.hashCode() : f2.hashCode();
  }
  Object m3() {
    return f4;
  }
}
```

Impacted Regions

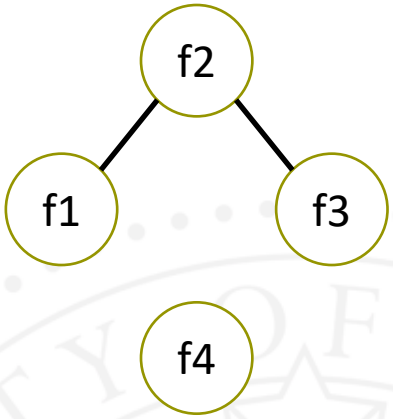
f1 -> m1

f2 -> {m1,m2}

f3 -> m2

f4 -> m3

Conflict graph



Optimization Example

```
class Test {
  Object f1 = null;
  Object f2 = null;
  Object f3 = null;
  Object f4 = null;
  Object f5 = f4;
  String m1() {
    return f1 != null ? f1.toString() : f2.toString();
  }
  int m2() {
    return f3 != null ? f3.hashCode() : f2.hashCode();
  }
  Object m3() {
    return f4;
  }
}
```

Impacted Regions

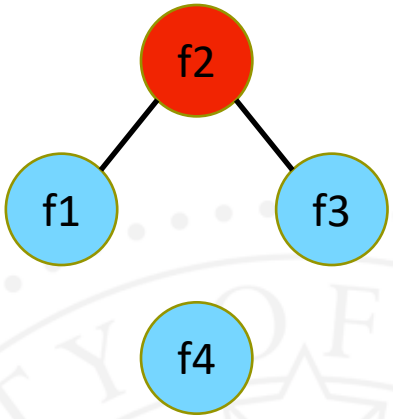
f1 -> m1

f2 -> {m1,m2}

f3 -> m2

f4 -> m3

Conflict graph



Results

- **Error reduction: 69.5% on average**
- **Average speedup 6.1X (2.0X – 17.8X)**
  - For two Uber targets, went from over 8 hours to less than 2 hours
- **6.32% unchecked code after suppressions**
  - Original model to suppress remaining warnings
- **Enrolled all Uber's Java services onto NullAway**

NullRepair (ISSTA'26)

Motivation

- Annotator leaves many warnings unaddressed
  - E.g., potential null dereferences
- Right fix for a null dereference is tricky
  - Could be a false positive
  - What to do in the default case?
  - **Wrong answer can be worse than a crash**

Solution: NullRepair

- Guided LLM-based approach to repairing NullAway warnings
- Key ideas
  - Leverage insights from a study of 200 errors and fixes
  - Use static analysis to give **safe usage region** context
- Improves over baseline agentic approach
  - Fewer warnings repaired, but
  - More tests passing
  - Better patches according to human evaluation

Example

```
 class User {
+  @Nullable MapView mapView;
+  @Nullable public MapView getMapView() {
     return this.mapView;
   }
 }
```

Example

```
 class User {
+  @Nullable MapView mapView;
+  @Nullable public MapView getMapView() {
     return this.mapView;
   }
 }
```

```
 class Controller {
   public void updateLocation(User user) {
-    move(user.getMapView().getCenter()); // Error
```

Example

```
 class User {
+  @Nullable MapView mapView;
+  @Nullable public MapView getMapView() {
     return this.mapView;
   }
 }
```

```
 class Controller {
   public void updateLocation(User user) {
-    move(user.getMapView().getCenter()); // Error
```

How to fix?

Example

```
public Point2D getCenter() {
  if (user.getMapView() == null) {
      return new Point2D.Double(-1, -1);
  }
  return user.getMapView().getCenter();
}
```

```
class Controller {
  public void updateLocation(User user) {
-   move(user.getMapView().getCenter()); // Error
```

Example

Safe usage region

```
public Point2D getCenter() {
  if (user.getMapView() == null) {
      return new Point2D.Double(-1, -1);
  }
  return user.getMapView().getCenter();
}
```

```
class Controller {
  public void updateLocation(User user) {
-   move(user.getMapView().getCenter()); // Error
```

Example

Safe usage region

```
public Point2D getCenter() {
  if (user.getMapView() == null) {
      return new Point2D.Double(-1, -1);
  }
  return user.getMapView().getCenter();
}
```

```
 class Controller {
   public void updateLocation(User user) {
-    move(user.getMapView().getCenter()); // Error
+    Point2D point = user.getMapView() == null  ?
+         new Point2D.Double(-1, -1)   :
+         user.getMapView().getCenter();
+    move(point);
   }
```

Key Steps

- Step 0: Run NullAway Annotator
- Step 1: Heuristic Rewrites
- Step 2: False Positive Suppression
- Step 3: Add More Annotations
- Step 4: LLM Repair with Safe Usage Regions

- **More details: Friday 1:30pm at ISSTA!**

False Positive Validation (Ongoing)

False Positive Suppression

- Sometimes, a suppression is the best “fix”
  - NullAway is imprecise for some cases
  - Unnecessary null check impairs readability
- NullRepair solution: ask LLM
  - Gives enclosing method, more context on demand
- **Problems: Correctness and Understandability**
  - LLM can label true positive as false
  - Suppression review can be challenging

Suppression Example

```
  @Nullable Session session;
  void handle() {
    if(check()) {  // 1
-     session.refresh(); // 2
+     Nullability.castToNonnull(session).refresh();
    }
  }
```

```
boolean check() {
  return session != null;
}
```

Suppression Example

```
  @Nullable Session session;
  void handle() {
    if(check()) {  // 1
-     session.refresh(); // 2
+     Nullability.castToNonnull(session).refresh();
    }
  }
```



```
boolean check() {
  return session != null;
}
```

Suppression Example

```
  @Nullable Session session;
  void handle() {
    if(check()) {  // 1
-     session.refresh(); // 2
+     Nullability.castToNonnull(session).refresh();
    }
  }
```



```
boolean check() {
  return session != null;
}
```

Structured reasoning

Suppression Example

```
  @Nullable Session session;
  void handle() {
    if(check()) {  // 1
-     session.refresh(); // 2
+     Nullability.castToNonnull(session).refresh();
    }
  }
```



```
boolean check() {
  return session != null;
}
```

```
1. method_postcondition
   METHOD: check
   WHEN: returns_true
   NON_NULL: this.session
```

Structured reasoning

Suppression Example

```
  @Nullable Session session;
  void handle() {
    if(check()) {  // 1
-     session.refresh(); // 2
+     Nullability.castToNonnull(session).refresh();
    }
  }
```



```
2. dominating_call
   METHOD: check
   CALL_SITE: 1
   DEREF: 2
```

```
boolean check() {
  return session != null;
}
```

```
1. method_postcondition
   METHOD: check
   WHEN: returns_true
   NON_NULL: this.session
```

Structured reasoning

Suppression Example

```
  @Nullable Session session;
  void handle() {
    if(check()) {  // 1
-     session.refresh(); // 2
+     Nullability.castToNonnull(session).refresh();
    }
  }
```



```
2. dominating_call
   METHOD: check
   CALL_SITE: 1
   DEREF: 2
```

```
boolean check() {
  return session != null;
}
```

```
1. method_postcondition
   METHOD: check
   WHEN: returns_true
   NON_NULL: this.session
```

Structured reasoning

Key Ideas

- **Constrain LLM output** to some fact language
  - Common fact types to prove deref safety
  - Ex: Postconditions, local dataflow, lifecycles
  - Escape hatch for facts that don't fit
- **Lightweight verifiers** to validate facts
  - Checker Framework, OpenJML
  - Our own targeted backwards symbolic executor
- **Iterate in agentic loop**
- **End result: validated false positives**
  - Aids review confidence
  - Facts can be persisted (e.g., in comment)

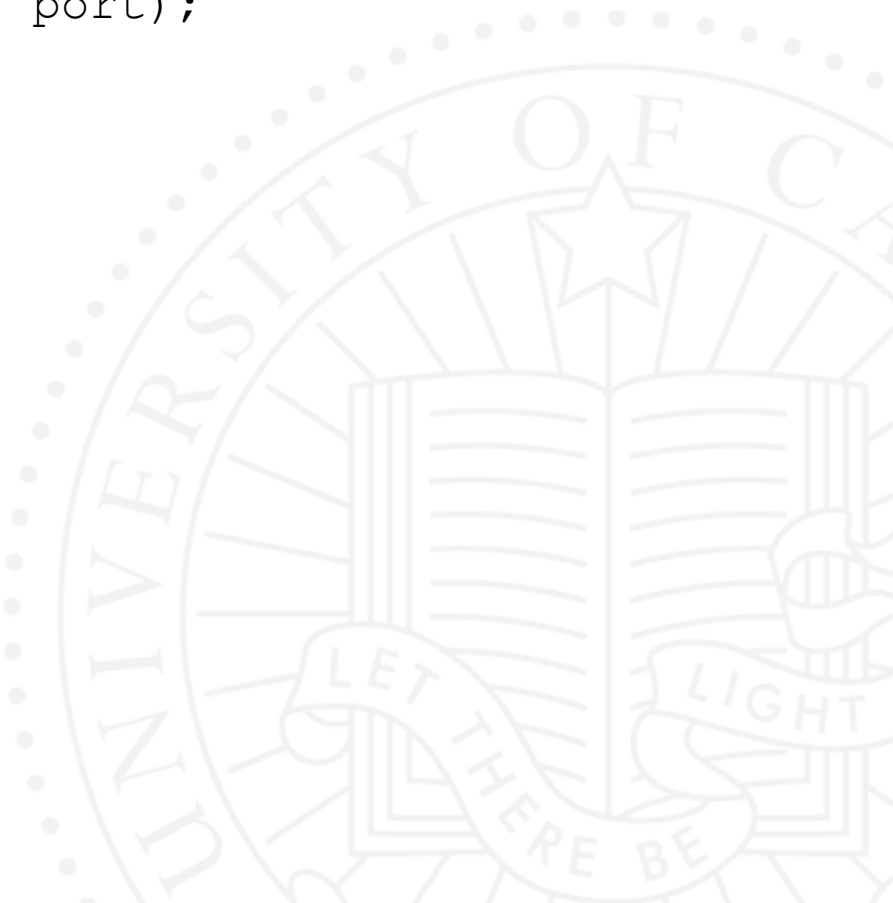
Resource Leak Inference (OOPSLA'23)



What's a Resource Leak?

```
try {
    Socket s = new Socket(address, port);
    ...
    s.close();
} catch (IOException e) {

}
```



What's a Resource Leak?

```
try {
    Socket s = new Socket(address, port);
    ...

} catch (IOException e) {


}
```



Missing call to close()

What's a Resource Leak?

```
try {
    Socket s = new Socket(address, port);
    ...
} catch (IOException e) {

}
```

Missing call to close()

Resources: Sockets, files, database connections, etc.

Resource Leak Problems

- Resource starvation
- Slowdowns
- System crashes
- Denial-of-service attack
  - E.g. CVE-1999-1127, CVE-2001-0830, CVE-2002-1372
- And, hard to test for!

Resource Leak Checker (FSE'21)

- **Sound(y), modular, incremental checker**
- **Insight:** can be sound without alias analysis
  - Accumulation analysis (ECOOP'22)
- **For precision:** targeted alias reasoning
  - Annotations expose key aliasing / ownership properties

RLC by example

```
class MySqlCon {
  private final         Connection con;

                MySqlCon(
                   Connection con) {
    this.con = con;
  }

  void use() { ... }


  void dispose() {
    closeCon(this.con);
  }

  static @Owning Connection createCon() {
    Connection obj = ...;
    return obj;
  }
```

```
  static void closeCon(@Owning Connection obj) {
    obj.close();
  }
} // end of class MySqlCon


static void client() {
  Connection con1 = MySqlCon.createCon();
  MySqlCon mySqlCon1 = new MySqlCon(con1);
  mySqlCon1.use();
  if (...)
    con1.close();
  else
    mySqlCon1.dispose();
}
```

RLC by example

```
class MySqlCon {
  private final         Connection con;

                MySqlCon(
                   Connection con) {
    this.con = con;
  }

  void use() { ... }


  void dispose() {
    closeCon(this.con);
  }

  static @Owning Connection createCon() {
    Connection obj = ...;
    return obj;
  }
```

```
  static void closeCon(@Owning Connection obj) {
    obj.close();
  }
} // end of class MySqlCon


static void client() {
  Connection con1 = MySqlCon.createCon();
  MySqlCon mySqlCon1 = new MySqlCon(con1);
  mySqlCon1.use();
  if (...)
    con1.close();
  else
    mySqlCon1.dispose();
}
```

RLC by example

```
class MySqlCon {
  private final         Connection con;

                MySqlCon(
                   Connection con) {
    this.con = con;
  }

  void use() { ... }


  void dispose() {
    closeCon(this.con);
  }

  static @Owning Connection createCon() {
    Connection obj = ...;
    return obj;
  }
```

```
  static void closeCon(@Owning Connection obj) {
    obj.close();
  }
} // end of class MySqlCon


  static void client() {
    Connection con1 = MySqlCon.createCon();
    MySqlCon mySqlCon1 = new MySqlCon(con1);
    mySqlCon1.use();
    if (...)
      con1.close();✅
    else
      mySqlCon1.dispose();
  }
```

RLC by example

```
class MySqlCon {
  private final         Connection con;

                MySqlCon(
                   Connection con) {
    this.con = con;
  }

  void use() { ... }


  void dispose() {
    closeCon(this.con);
  }

  static @Owning Connection createCon() {
    Connection obj = ...;
    return obj;
  }
```

```
  static void closeCon(@Owning Connection obj) {
    obj.close();
  }
} // end of class MySqlCon


static void client() {
  Connection con1 = MySqlCon.createCon();
  MySqlCon mySqlCon1 = new MySqlCon(con1);
  mySqlCon1.use();
  if (...)
    con1.close();✅
  else
    mySqlCon1.dispose();
}
```

RLC by example

```
@MustCall("dispose")
class MySqlCon {
  private final @Owning Connection con;

  @ResourceAlias MySqlCon(
     @ResourceAlias Connection con) {
    this.con = con;
  }

  void use() { ... }


  void dispose() {
    closeCon(this.con);
  }

  static @Owning Connection createCon() {
    Connection obj = ...;
    return obj;
  }

  static void closeCon(@Owning Connection obj) {
    obj.close();
  }
} // end of class MySqlCon


static void client() {
  Connection con1 = MySqlCon.createCon();
  MySqlCon mySqlCon1 = new MySqlCon(con1);
  mySqlCon1.use();
  if (...)
    con1.close();✅
  else
    mySqlCon1.dispose();
}
```

RLC by example

```
@MustCall("dispose")
class MySqlCon {
  private final @Owning Connection con;

  @ResourceAlias MySqlCon(
     @ResourceAlias Connection con) {
    this.con = con;
  }

  void use() { ... }


  void dispose() {
    closeCon(this.con);
  }

  static @Owning Connection createCon() {
    Connection obj = ...;
    return obj;
  }

  static void closeCon(@Owning Connection obj) {
    obj.close();
  }
} // end of class MySqlCon


static void client() {
  Connection con1 = MySqlCon.createCon();
  MySqlCon mySqlCon1 = new MySqlCon(con1);
  mySqlCon1.use();
  if (...)
    con1.close();✅
  else
    mySqlCon1.dispose();✅
}
```

RLC by example

```
@MustCall("dispose")
class MySqlCon {
  private final @Owning Connection con;

  @ResourceAlias MySqlCon(
     @ResourceAlias Connection con) {
    this.con = con;
  }

  void use() { ... }

  @Calls("this.con", "close")
  void dispose() {
    closeCon(this.con);
  }

  static @Owning Connection createCon() {
    Connection obj = ...;
    return obj;
  }

  static void closeCon(@Owning Connection obj) {
    obj.close();
  }
} // end of class MySqlCon


static void client() {
  Connection con1 = MySqlCon.createCon();
  MySqlCon mySqlCon1 = new MySqlCon(con1);
  mySqlCon1.use();
  if (...)
    con1.close();✅
  else
    mySqlCon1.dispose();✅
}
```

RLC by example

```
@MustCall("dispose")
class MySqlCon { ✅
  private final @Owning Connection con;

  @ResourceAlias MySqlCon(
     @ResourceAlias Connection con) {
    this.con = con;
  }

  void use() { ... }

  @Calls("this.con", "close")
  void dispose() {
    closeCon(this.con);
  }

  static @Owning Connection createCon() {
    Connection obj = ...;
    return obj;
  }

  static void closeCon(@Owning Connection obj) {
    obj.close();
  }
} // end of class MySqlCon
```

```
static void client() {
  Connection con1 = MySqlCon.createCon();
  MySqlCon mySqlCon1 = new MySqlCon(con1);
  mySqlCon1.use();
  if (...)
    con1.close(); ✅
  else
    mySqlCon1.dispose(); ✅
}
```

RLC by example

```
@MustCall("dispose")
class MySqlCon { ✅
  private final @Owning Connection con;

  @ResourceAlias MySqlCon(
     @ResourceAlias Connection con) {
    this.con = con;
  }

  void use() { ... }

  @Calls("this.con", "close")
  void dispose() {
    closeCon(this.con);
  }

  static @Owning Connection createCon() {
    Connection obj = ...;
    return obj;
  }

  static void closeCon(@Owning Connection obj) {
    obj.close();
  }
} // end of class MySqlCon
```



```
static void client() {
  Connection con1 = MySqlCon.createCon();
  MySqlCon mySqlCon1 = new MySqlCon(con1);
  mySqlCon1.use();
  if (...)
    con1.close(); ✅
  else
    mySqlCon1.dispose(); ✅
}
```

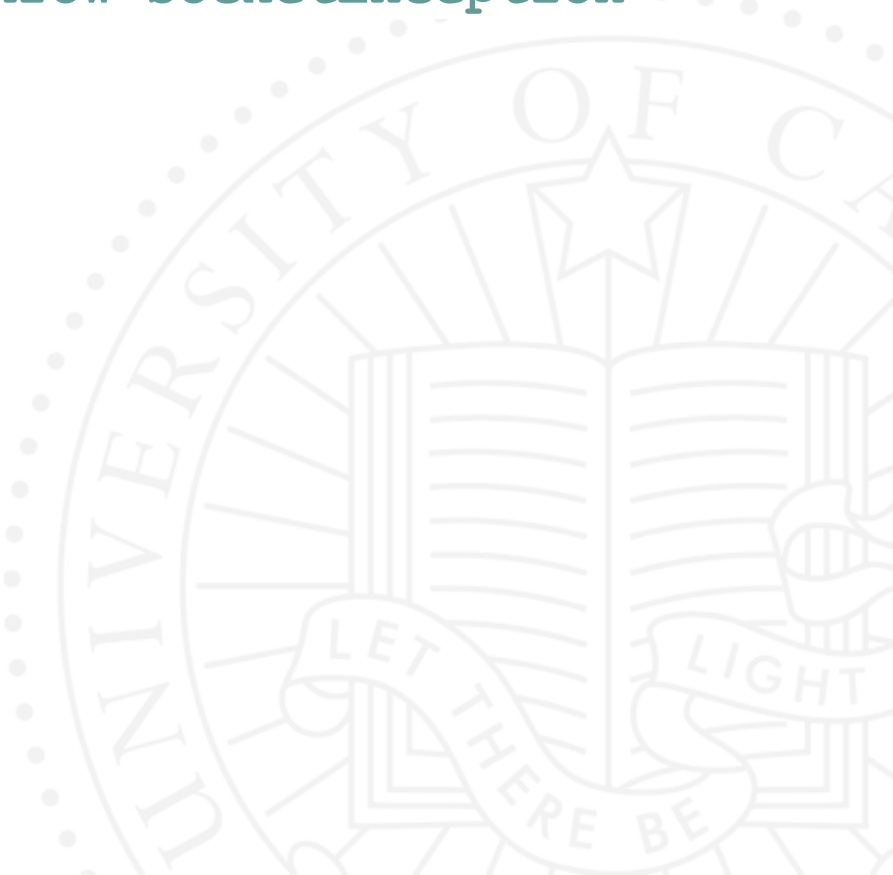
Inference needed!

Inference Overview

- (Inter-related) resource specification properties require a custom approach:
  - Resource ownership
  - Call obligations
  - Resource aliasing relationships
- Algorithm is *optimistic*: infer specs aligning with developer's *likely intention*
- Formalized as inference rules, applied to fixed point
- Two implementations:
  - Checker Framework for Java
  - CodeQL for C#

Optimistic Inference Example

```
void cleanup(Socket socket) throws SocketException {
  socket.shutdown(); // may throw SocketException
  socket.close();
}
```



Optimistic Inference Example

```
void cleanup(Socket socket) throws SocketException {
  socket.shutdown(); // may throw SocketException
  socket.close();
}
```

- `cleanup` may not close socket



Optimistic Inference Example

```
void cleanup(Socket socket) throws SocketException {
  socket.shutdown(); // may throw SocketException
  socket.close();
}
```

- `cleanup` may not close socket
- Standard inference: only capture code behavior
  - No **`@Owning`** annotation for `socket`
  - May lead to false positives at calls to `cleanup`

Optimistic Inference Example

```
void cleanup(@Owning Socket socket) throws … {
  socket.shutdown(); // may throw SocketException
  socket.close();
}
```

- `cleanup` may not close socket
- Standard inference: only capture code behavior
  - No **`@Owning`** annotation for `socket`
  - May lead to false positives at calls to `cleanup`
- Optimistic: try to *mirror developer intent*
  - Infer **`@Owning`** for **`socket`**
  - RLC issues error in method, where bug fix is needed

Results: Recovering Manual Specs

	Hand-written Specifications	Inferred Specifications	Percentage (%)
Service 1	21	21	100
Service 2	28	28	100
Service 3	24	24	100
Lucene.Net	63	60	95
EF Core	25	17	68
Zookeeper	93	66	71
Hadoop-hdfs	91	69	76
Hbase	35	26	74

- Recovery rate: 82% for open-source C#, 74% for open-source Java, 100% for proprietary C#

Results: Impact on Warnings

NS: No Specifications IS: Inferred Specifications

	#warnings (NS)	#warnings (IS)
Service 1	251	240
Service 2	45	34
Service 3	20	12
Lucene.Net	670	592
EF Core	88	147
Zookeeper	138	170
Hadoop-hdfs	26	95
Hbase	828	844

Results: Impact on Warnings

NS: No Specifications IS: Inferred Specifications

	#warnings (NS)	#warnings (IS)
Service 1	251	240
Service 2	45	34
Service 3	20	12
Lucene.Net	670	592
EF Core	88	147
Zookeeper	138	170
Hadoop-hdfs	26	95
Hbase	828	844

Inferred annotations lead to new warnings!

Results: Impact on Warnings

```
1  @MustCall("dispose")
2  class MySqlCon {
3    private final @Owning Connection con;
4
5    @ResourceAlias MySqlCon(
6       @ResourceAlias Connection con) {
7      this.con = con;
8    }
9
10   void use() { ... }
```

Results: Impact on Warnings

Will lead to warnings at uses of `MySqlCon`

```
1  @MustCall("dispose")
2  class MySqlCon {
3    private final @Owning Connection con;
4
5    @ResourceAlias MySqlCon(
6        @ResourceAlias Connection con) {
7      this.con = con;
8    }
9
10   void use() { ... }
```

Results: Impact on Warnings II

	True Positives	Incorrect Specifications	Missing Specifications
C#	28%	0.4%	25%
Java	19%	5%	27%

True positive rate similar to hand-written specs

Results: Impact on Warnings II

	True Positives	Incorrect Specifications	Missing Specifications
C#	28%	0.4%	25%
Java	19%	5%	27%

Few warnings due to incorrect specs

Results: Impact on Warnings II

	True Positives	Incorrect Specifications	Missing Specifications
C#	28%	0.4%	25%
Java	19%	5%	27%

Some warnings due to missing specs
(Common causes: non-resource types in C#, mutable fields in Java)

Resource Leak Repair (ASE'25)

Motivation

- Like nullness, warnings remain after annotating
  - E.g., due to real leaks
- RLFixer (Utture and Palsberg, FSE'23)
  - Escape analysis to determine safe close point
    - Give up on escapes to fields / data structures
  - Template-based fix patterns
  - Some reasoning about resource aliases
  - Analyzer-independent: tested with RLC
- **Key gap:** user-defined wrapper types

RLFixer Fails for Wrappers

```
class TempWriter {
  private PrintStream stream;
  TempWriter(String path) {
    stream = new PrintStream(path);
  }
  public void printSomething() {
    stream.println("hello");
  }
}
```

- **Black-box model:** *Detector → Repair (RLFixer)*

RLFixer Fails for Wrappers

```
class TempWriter {
  private PrintStream stream;
  TempWriter(String path) {
    stream = new PrintStream(path);
  }
  public void printSomething() {
    stream.println("hello");
  }
}
```

- **Black-box model:** *Detector → Repair (RLFixer)*
- **Detector warns inside wrapper → wrong location** for repair

RLFixer Fails for Wrappers

```
class TempWriter {
  private PrintStream stream;
  TempWriter(String path) {
    stream = new PrintStream(path);
  }
  public void printSomething() {
    stream.println("hello");
  }
}
```

- **Black-box model:** *Detector → Repair (RLFixer)*
- **Detector warns inside wrapper → wrong location** for repair
- **Resource in field** → repair tool says “might escape” → **skip**

RLFixer Fails for Wrappers

```
class TempWriter {
  private PrintStream stream;
  TempWriter(String path) {
    stream = new PrintStream(path);
  }
  public void printSomething() {
    stream.println("hello");
  }
}
```

- **Black-box model:** *Detector → Repair (RLFixer)*
- **Detector warns inside wrapper → wrong location** for repair
- **Resource in field** → repair tool says “might escape” → **skip**
- **Prior work:** 41% cases repaired. Of what remained, ≈62% were field-held.

RLFixer Fails for Wrappers

```
class TempWriter {
  private PrintStream stream;
  TempWriter(String path) {
    stream = new PrintStream(path);
  }
  public void printSomething() {
    stream.println("hello");
  }
}
```

Incomplete wrapper: missing disposal method, so RLC inference fails

- **Black-box model:** *Detector → Repair (RLFixer)*
- **Detector warns inside wrapper → wrong location** for repair
- **Resource in field** → repair tool says “might escape” → **skip**
- **Prior work:** 41% cases repaired. Of what remained, ≈62% were field-held.

Arodnap Example

```
class TempWriter {
  private PrintStream stream;
  public TempWriter(String path) {
    stream = new PrintStream(path);
  }
  public void printSomething() {
    stream.println("hello");
  }
}
```

Detect → Transform → Infer → Re-detect → Repair

Arodnap Example

```
class TempWriter {
  private PrintStream stream;
  public TempWriter(String path) {
    stream = new PrintStream(path);
  }
  public void printSomething() {
    stream.println("hello");
  }
}
```

Detected Leak

Detect → Transform → Infer → Re-detect → Repair

Arodnap Example

```
class TempWriter {
  private PrintStream stream;
  public TempWriter(String path) {
    stream = new PrintStream(path);
  }
  public void printSomething() {
    stream.println("hello");
  }
}
```

Detected Leak

Use-after-close

Detect → Transform → Infer → Re-detect → Repair

Arodnap Example

```
class TempWriter implements AutoCloseable {
  private PrintStream stream;
  public TempWriter(String path) {
    stream = new PrintStream(path);
  }
  public void printSomething() {
    stream.println("hello");
  }
  public void close() {
    stream.close();
  }
}
```

Detect → **Transform** → Infer → Re-detect → Repair

Arodnap Example

```
class TempWriter implements AutoCloseable {
  @Owning private PrintStream stream;
  public TempWriter(String path) {
    stream = new PrintStream(path);
  }
  public void printSomething() {
    stream.println("hello");
  }
  public void close() {
    stream.close();
  }
}
```

Inference identifies **TempWriter** is responsible for closing the resource

Detect → Transform → **Infer** → Re-detect → Repair

Arodnap Example

```
class TempWriter implements AutoCloseable {
  @Owning private PrintStream stream;
  public TempWriter(String path) {
    stream = new PrintStream(path);
  }
  public void printSomething() {
    stream.println("hello");
  }
  public void close() {
    stream.close();
  }
}
```

```
void client() {
  TempWriter w = new TempWriter("f.txt");
  w.printSomething();
}

void client2() {...}
```

Detect → Transform → Infer → **Re-detect** → Repair

Arodnap Example

```
class TempWriter implements AutoCloseable {
  @Owning private PrintStream stream;
  public TempWriter(String path) {
    stream = new PrintStream(path);
  }
  public void printSomething() {
    stream.println("hello");
  }
  public void close() {
    stream.close();
  }
}
```

```
void client() {
  TempWriter w = null;
  try {
    w = new TempWriter("f.txt");
    w.printSomething();
  } finally {
    if (w != null) {
      w.close();
    }
  }
}
```

Detect → Transform → Infer → Re-detect → **Repair**

Hard-to-Repair Field-Held Leaks

```
class TempWriter implements AutoCloseable {
  @Owning private PrintStream stream;
  public TempWriter(String path) {
    stream = new PrintStream(path);
  }
  public void close() {
     stream.close();
  }



}
```

Hard-to-Repair Field-Held Leaks

```
class TempWriter implements AutoCloseable {
  @Owning private PrintStream stream;
  public TempWriter(String path) {
    stream = new PrintStream(path);
  }
  public void close() {
     stream.close();
  }
  public void resetStream(String path) {
    stream = new PrintStream(path);
  }
}
```

Hard-to-Repair Field-Held Leaks

```
class TempWriter implements AutoCloseable {
  @Owning private PrintStream stream;
  public TempWriter(String path) {
    stream = new PrintStream(path);
  }
  public void close() {
     stream.close();
  }
  public void resetStream(String path) {
    stream = new PrintStream(path);
  }
}
```

Leak due to reassignment

Hard-to-Repair Field-Held Leaks

```
class TempWriter implements AutoCloseable {
  @Owning private PrintStream stream;
  public TempWriter(String path) {
    stream = new PrintStream(path);
  }
  public void close() {
     stream.close();
  }
  public void resetStream(String path) {
    stream = new PrintStream(path);
  }
}
```

Field Containment Analysis

Leak due to reassignment

Hard-to-Repair Field-Held Leaks

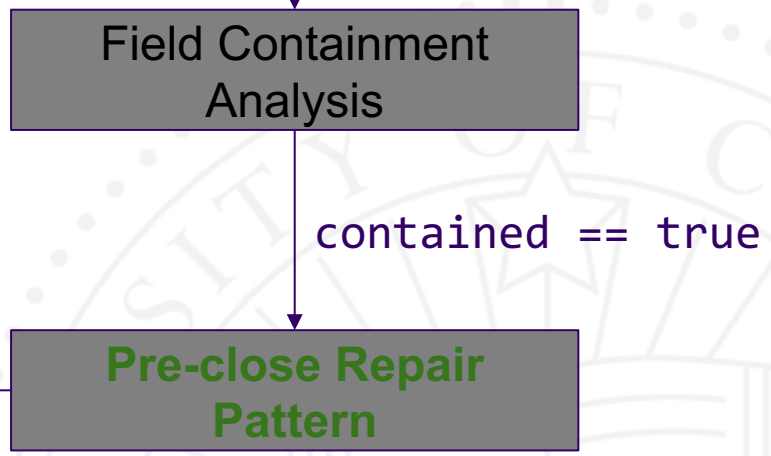
```
class TempWriter implements AutoCloseable {
  @Owning private PrintStream stream;
  public TempWriter(String path) {
    stream = new PrintStream(path);
  }
  public void close() {
     stream.close();
  }
  public void resetStream(String path) {
    stream = new PrintStream(path);
  }
}
```

Field Containment Analysis

Leak due to reassignment

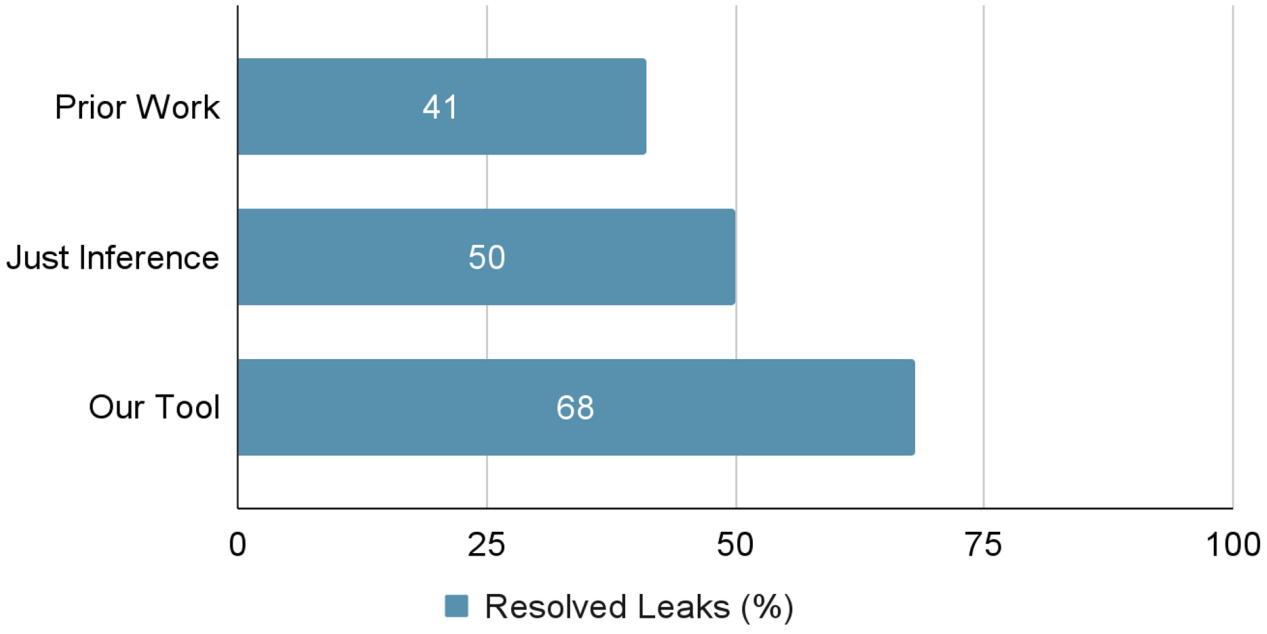
Hard-to-Repair Field-Held Leaks

```
class TempWriter implements AutoCloseable {
  @Owning private PrintStream stream;
  public TempWriter(String path) {
    stream = new PrintStream(path);
  }
  public void close() {
     stream.close();
  }
  public void resetStream(String path) {
    if (stream != null) stream.close();
    stream = new PrintStream(path);
  }
}
```



Arodnap Resolves 68% of Leaks

- **Dataset:** 285 open-source Java projects (NJR).



Future Work

- Richer properties for checking / inference / repair
  - Extending RLC to handle resource collections
  - Simple dependent types with boolean fields
- A unified inference / repair approach across pluggable type systems?
  - Still grounded in static analysis / verification
- Type error messages to better help agents
  - E.g., include details from dataflow stores

Conclusions

- **Pluggable Types + Inference + Repair**
  - Stricter checking provides big value with agents
  - Inference + repair helps to migrate existing code
- **Two instantiations: nullness and resource leaks**
- **Let's bring more verification to everyone!**

Thanks! Questions?

NullAway:
https://github.com/uber/NullAway

NullAway Annotator:
https://github.com/ucr-riple/NullAwayAnnotator

NullRepair: https://github.com/Pascal-Joos/NullRepair

RLC + inference:
https://checkerframework.org/

Arodnap:
https://github.com/typetools/arodnap



Bonus

Inference Rule Example

```
void cleanup(@Owning Socket socket) throws … {
  socket.shutdown(); // may throw SocketException
  socket.close();
}
```

Inference Rule Example

```
void cleanup(@Owning Socket socket) throws … {
  socket.shutdown(); // may throw SocketException
  socket.close();
}
```

$$\textbf{ParamAnnot}(\texttt{Owning}, m, p) \Leftarrow \\ \quad \text{ParamType}(m, T), \\ \quad \textbf{ClassAnnot}(\texttt{MustCall}(m_{pd}), T), \\ \quad \text{Invokes}(s, m, m_{pd}, p, _)$$

Inference Rule Example

```
void cleanup(@Owning Socket socket) throws … {
  socket.shutdown(); // may throw SocketException
  socket.close();
}
```

ParamAnnot(`Owning`, m, p) $\Leftarrow$
  ParamType(m, T),
  ClassAnnot(`MustCall`(m_{pd}), T),
  Invokes(s, m, m_{pd}, p, _)

Type T of `socket` is `Socket`

Inference Rule Example

```
void cleanup(@Owning Socket socket) throws … {
  socket.shutdown(); // may throw SocketException
  socket.close();
}
```

ParamAnnot(`Owning`, m, p) $\Leftarrow$
  ParamType(m, T),
  ClassAnnot(`MustCall`(m_{pd}), T),
  Invokes(s, m, m_{pd}, p, _)

We model `Socket` as `@MustCall("close")`

Inference Rule Example

```
void cleanup(@Owning Socket socket) throws … {
  socket.shutdown(); // may throw SocketException
  socket.close();
}
```

$$\textbf{ParamAnnot}(\texttt{Owning}, m, p) \Leftarrow \\ \quad \text{ParamType}(m, T), \\ \quad \textbf{ClassAnnot}(\texttt{MustCall}(m_{pd}), T), \\ \quad \text{Invokes}(s, m, m_{pd}, p, _)$$

There exists (optimistic!) a call to `close` with receiver `socket` within `cleanup`