# A New Approach to Evaluating Nullability Inference Tools

NIMA KARIMIPOUR*, University of California, Riverside, USA
ERFAN ARVAN*, New Jersey Institute of Technology, USA
MARTIN KELLOGG, New Jersey Institute of Technology, USA
MANU SRIDHARAN, University of California, Riverside, USA

Null-pointer exceptions are serious problem for Java, and researchers have developed type-based nullness checking tools to prevent them. These tools, however, have a downside: they require developers to write nullability annotations, which is time-consuming and hinders adoption. Researchers have therefore proposed *nullability annotation inference tools*, whose goal is to (partially) automate the task of annotating a program for nullability. However, prior works rely on differing theories of what makes a set of nullability annotations good, making comparing their effectiveness challenging.

In this work, we identify a systematic bias in some prior experimental evaluation of these tools: the use of "type reconstruction" experiments to see if a tool can recover erased developer-written annotations. We show that developers make semantic code changes while adding annotations to facilitate typechecking, leading such experiments to overestimate the effectiveness of inference tools on never-annotated code. We propose a new definition of the "best" inferred annotations for a program that avoids this bias, based on a systematic exploration of the design space. With this new definition, we perform the first head-to-head comparison of three extant nullability inference tools. Our evaluation showed the complementary strengths of the tools and remaining weaknesses that could be addressed in future work.

CCS Concepts: • **Software and its engineering**;

Additional Key Words and Phrases: pluggable type systems, null saftey, inference, inference evaluation

**ACM Reference Format:**
Nima Karimipour, Erfan Arvan, Martin Kellogg, and Manu Sridharan. 2025. A New Approach to Evaluating Nullability Inference Tools. *Proc. ACM Softw. Eng.* 2, FSE, Article FSE017 (July 2025), 22 pages. https://doi.org/10.1145/3715732

## 1 Introduction

Null-pointer exceptions are still a serious problem in Java, despite decades of research [Banerjee et al. 2019; Dietl et al. 2011; Hovemeyer et al. 2005; Karimipour et al. 2023; Loginov et al. 2008; Madhavan and Komondoor 2011; Papi et al. 2008; Siddiqui and Kellogg 2024]. Static analysis is a popular way to detect possible null-pointer exceptions before they happen. *Nullability annotations* like `@Nullable` and `@NonNull` can improve the effectiveness of common nullability checkers. Further, these annotations are required to get the full benefit of tools that not only detect some null-pointer exceptions but also claim to detect most or all possible null-pointer exceptions, such as pluggable

*Both authors contributed equally to this work.

Authors' Contact Information: Nima Karimipour, nima.karimipour@email.ucr.edu, University of California, Riverside, Riverside, California, USA; Erfan Arvan, ea442@njit.edu, New Jersey Institute of Technology, Newark, NJ, USA; Martin Kellogg, martin.kellogg@njit.edu, New Jersey Institute of Technology, Newark, NJ, USA; Manu Sridharan, manu@cs.ucr.edu, University of California, Riverside, Riverside, California, USA.

ACM 2994-970X/2025/7-ARTFSE017
https://doi.org/10.1145/3715732

typecheckers like Uber's NullAway [Banerjee et al. 2019], Meta's Nullsafe [Pianykh et al. 2022], or the Checker Framework's Nullness Checker ("CFNullness") [Dietl et al. 2011; Papi et al. 2008].

Several recent works [Karimipour et al. 2023; Kellogg et al. 2023; Siddiqui and Kellogg 2024] have proposed static *nullability annotation inference techniques* that take as input a program without nullability annotations and output a set of inferred annotations for that program. However, each technique is based on a different underlying inference strategy:

- Nullaway Annotator [Karimipour et al. 2023] treats a nullability checker as a black box fitness oracle: it places annotations that minimize the total number of warnings from the checker.
- Checker Framework Whole-Program Inference (WPI) [Kellogg et al. 2023] bootstraps its inference from the local dataflow analysis that CFNullness already performs during typechecking.
- NullGTN [Siddiqui and Kellogg 2024] uses a deep-learning model trained on open-source code from GitHub to place nullability annotations in the same places where its model predicts that humans would.

Each technique relies on a different theory of which annotations are desirable. Nullaway Annotator assumes that the best annotations *minimize errors from a checking tool.* WPI assumes that the best annotations *encode which program locations might actually be null.* NullGTN assumes that the best annotations are *similar to those that humans write.* These differing assumptions lead to inference tools that may behave quite differently in practice. This work aims to explore the design space of nullability inference tools and formalize what makes a set of annotations "good."

The papers introducing these techniques evaluated each in isolation. Our work contains the first head-to-head comparision between them on the same large set of benchmark programs. Designing a fair head-to-head comparison between these tools to properly evaluate their usefulness is not simple. The key difficulty is that the envisioned deployment scenario for such a tool is annotating a program that *has not been checked by a nullability detection tool.* When evaluating an inference tool, though, an experimenter desires *ground-truth annotations*: that is, annotations that are known to be correct, which the experimenter can use to evaluate the quality of the tool's annotations. Therefore, some prior works have done *type reconstruction* experiments, wherein a tool is evaluated by 1) removing the ground-truth annotations written by a human, 2) running the tool to produce a set of annotations, and 3) comparing the tool-generated annotations to the ground-truth annotations.

The type reconstruction experimental design assumes that developers do not make changes to their code as they add annotations, and that using a nullability checking tool does not impact coding style. These assumptions are not obviously true: e.g., developers could make code changes to suppress false positive tool warnings [van Tonder and Le Goues 2020] or fix real bugs exposed by the tool. So, type reconstruction experiments could introduce a *systematic bias* in the evaluation: inference for code that has been annotated by a human could be easier than inference for code that has never been annotated. We show empirically in section 3 that this systematic bias exists for nullability inference: developers make other semantic changes to code as they annotate, and these changes make both checking and inference more effective. Evaluating a nullability inference tool on a previously-annotated benchmark therefore *overestimates* its usefulness for its true task of reducing developer workload when annotating code that has not been annotated before. While we only focus on nullability inference, the same bias may exist in other kinds of type inference experiments (see section 3.3.1).

It is not simple to correct this bias and fairly compare inference tools on benchmarks with no ground-truth annotations. We address this problem by constructing a model of an idealized inference tool and defining the "best" nullability annotations for a program, based on a set of principles for what makes an annotation "good." While these principles do not fully capture developer intent, they are useful guidelines for evaluating annotation quality. We then ran each inference tool on a large dataset of previously-unannotated Java programs, and used our definition to 1) manually evaluate the quality of (a sample of) each tool's annotations, and 2) define two automatically-measurable proxies that give more insight into each tool's performance. Our results show that while the three studied tools are complementary, the Nullaway Annotator tool produces the best output overall. Nevertheless, we conclude that no tool fully solves the problem of inferring "good" nullability annotations, and point to promising directions for future work. In summary, our contributions are:

- experimental confirmation that type reconstruction experiments overestimate the effectiveness of nullability annotation inference tools (section 3);
- an exploration of the design space of nullability annotation inference tools and evaluations, with an emphasis on how to fairly evaluate these tools in the absence of ground-truth annotations (section 4);
- an evaluation of three extant nullability annotation inference tools using a new methodology that avoids the bias inherent in annotation reconstruction experiments (section 5).

## 2 Background: Design Space of Extant Inference Tools

This section explains the design philosophies of the studied nullability annotation inference tools:

- Checker Framework Whole-Program Inference (WPI) [Kellogg et al. 2023], whose design philosophy emphasizes inferring *annotations that can be derived via dataflow analysis*;
- Nullaway Annotator [Karimipour et al. 2023], whose design philosophy emphasizes inferring *annotations that minimize a checking tool's warnings*; and
- NullGTN [Siddiqui and Kellogg 2024], whose design philosophy emphasizes inferring the *annotations that a human would have written.*

After discussing the extant tools, we discuss how the tools were evaluated, with a focus on to what extent the evaluation of each tool in prior work is impacted by reliance on type reconstruction experiments (section 2.4); we assess the potential bias in such experiments in section 3.

### 2.1 Checker Framework WPI

Checker Framework Whole-Program Inference (WPI) [Kellogg et al. 2023] is a type inference algorithm designed to infer type qualifiers for any pluggable type system. WPI operates at the typechecking-framework level and leverages the existing local inference capabilities of the type-checkers to derive a global inference algorithm. WPI iteratively uses locally inferred types to generate method, class, and field qualifiers, which are then refined through subsequent iterations to a fixed point.

The animating design philosophy of WPI is *reusing dataflow analysis work that a typechecker is already doing* to infer annotations. For nullness, WPI infers annotations based on the expression nullability facts computed by CFNullness. For example, if an expression assigned to a field might be nullable, WPI will infer that the field should have an `@Nullable` annotation. WPI supports all of the annotations supported by CFNullness (not just `@Nullable`), including pre- and post-condition annotations on methods, initialization annotations on method parameters, and monotonically non-null fields [CF Developers 2024]. WPI inherits the conservatism of CFNullness: if CFNullness's dataflow analysis suggests that any flow into a program element is possibly nullable, then WPI will always make that program element `@Nullable`.

Figure 1 is an example that WPI handles well. The `data` field is initialized lazily; code should call the `lazyInit` method before dereferencing it. WPI correctly infers a `@Nullable` annotation for the field. It can also infer an `@EnsuresNonNull` annotation for `lazyInit`, capturing the fact that `data` is always non-null when the method returns; this annotation lets CFNullness verify the code.

```
class DataHandler {
  +@Nullable Data data = null;
  +@EnsuresNonNull(expr={"this.data"})
  void lazyInit() {
    if (data == null)
      data = new Data(...);
  }
  String serialize() {
    lazyInit();
    return data.toString(); // safe
  }
}
```

**Fig. 1: Example where WPI correctly infers nullability annotations.**

```
class MyActivity {
  Name name;
  +@Nullable Address addr;
  MyActivity() {
    name = null; addr = null;
  }
  @Initializer
  void onStart() { name = defaultName(); }
  void doFirst() { name.showFirst(); }
  void doLast() {
    name.showLast();
    if (addr != null) addr.show();
  }
}
```

**Fig. 2: Example where Nullaway Annotator correctly infers nullability annotations.**

## 2.2 Nullaway Annotator

Nullaway Annotator (Annotator) [Karimipour et al. 2023] is a tool to infer annotations required to enroll programs into NullAway checking [Banerjee et al. 2019]. Annotator aims to infer a set of `@Nullable` annotations that minimizes the number of errors reported by NullAway. Treating NullAway as a black box, Annotator computes annotations to locally resolve each reported NullAway error. Since applying these annotations may lead to new NullAway errors, Annotator uses a backtracking search to explore sets of annotation fixes to some depth limit, in the end retaining the fixes that reduced NullAway errors the most.

The hypothesis behind Annotator's approach is that if adding a `@Nullable` annotation causes more errors than it resolves and those new errors cannot also be resolved by adding more annotations, it may reflect a contradiction in the program's assumptions. In such cases, developers may prefer not to add `@Nullable` and instead address the issue via a different annotation, rewriting the code, or a warning suppression (if the code is correct but cannot be verified). However, this hypothesis was *not* experimentally evaluated by the tool's authors [Karimipour et al. 2023]. The key difference between Annotator and WPI is that Annotator does not always encode the nullability of a program element, as judged by the underlying checker, into an annotation due to this error minimization algorithm.

Figure 2 gives an example that mimicks the structure of Android Activity classes [Google 2024a] that Annotator handles well. The `name` and `addr` fields are both initialized to `null` in the constructor. However, `name` is initialized in the `onStart` method, which runs early in the Android activity lifecycle [Google 2024b]. Hence, `doFirst` and `doLast` can rely on `name` being initialized and dereference it without a null check. Annotator's heuristic to minimize errors correctly handles this case, as making `name` `@Nullable` removes one error in the constructor but creates errors at the two dereferences. NullAway supports the `@Initializer` annotation on `onStart` to capture its intended usage and remove the constructor error (though Annotator does not infer this `@Initializer` annotation, it can in other cases [Karimipour et al. 2023]). Annotator correctly annotates `addr` as `@Nullable`. Note Annotator's error minimization technique does *not* lead to the desired result for fig. 1. Annotator sees that making `data` `@Nullable` increases the error count and avoids it. But this approach hides errors if a `lazyInit` call were missing before a use of `data`. In contrast, WPI's technique does not work as desired for fig. 2; it makes `name` `@Nullable`, when it cannot be `null` at its uses due to initialization in `onStart`. Our work is the first to study how this tradeoff impacts results on real benchmarks.

## 2.3 NullGTN

NullGTN [Siddiqui and Kellogg 2024] is machine-learning-based nullability inference tool based on the Graph Transformer Network (GTN) architecture. It uses a novel representation called NaP-AST (Name-augmented, Pruned Abstract Syntax Tree), which encodes minimal dataflow hints to enhance inference.

The key principle of NullGTN's design is *emulating the places that humans would write nullability annotations*, without regard for the semantics of a checking tool. This principle informs the data used to train NullGTN: any code that contained a nullability annotation was included, whether or not the code passed a nullness checking tool. The authors did not compute exactly how much of their training data came from code annotated for a checking tool, as opposed to just for documentation. But, given that use of type-based nullness checking tools is not yet widespread, and that the dataset contains over 32,000 classes, it is reasonable to infer that a significant portion of their data set is code annotated primarily for documentation purposes.

```
public int sumLengths(int +@Nullable [] v1,
  int +@Nullable [] v2) {
 int result = 0;
 result += (v1 == null) ? 0 : v1.length;
 result += (v2 == null) ? 0 : v2.length;
 return result; }
```

**Fig. 3: Example where NullGTN correctly infers annotations.**

Unlike the other tools, NullGTN can annotate the parameters of entrypoint methods—that is, methods intended to be called from outside the program—even if the program under analysis contains no calls to those methods. Consider the library method (based on one from our study) in fig. 3. It is clearly safe to pass `null` parameters to `sumLengths`. NullGTN can infer `@Nullable` annotations for the parameters without observing any call sites to `sumLengths`, while WPI and Annotator must observe calls passing `null` to infer these annotations. Such entrypoint APIs were the single largest cause for WPI to fail to infer annotations in the original WPI paper's experiments [Kellogg et al. 2023, Table III], and in this paper we study the impact of this potential advantage in practice.

## 2.4 Prior Evaluations

*2.4.1 Checker Framework WPI.* WPI was evaluated on 12 previously-annotated, open-source projects that each typechecked with at least one pluggable typechecker (11 of them with CFNullness). The evaluation used *type reconstruction experiments*: the authors removed the human-written annotations in each benchmark, ran their inference tool (in this case WPI), and compared the inferred annotations to the human-written annotations. Specifically, they measured the number of inferred annotations, the percentage of human-written annotations WPI could infer, and the number of typechecking errors before and after inference. The results showed that WPI inferred 39% of human-written annotations on average and reduced typechecking warnings by 45%.

*2.4.2 Nullaway Annotator.* In the evaluation of Nullaway Annotator, the authors used 14 open-source Java projects from GitHub that had not been annotated for nullability before. The key metrics for evaluation were the number of NullAway errors after inference, the percentage reduction in errors, and the performance of the tool (i.e., run time). In its best configuration, Annotator averaged an error reduction of 69.5% across the open-source benchmarks.

*2.4.3 NullGTN.* The evaluation of NullGTN used 12 open-source projects from the original NullAway [Banerjee et al. 2019] study, which were previously annotated for nullability; the NullGTN model was compared to two other candidate machine-learning models. Each project was checked with NullAway in three configurations: with the original human-written annotations, after removing all human-written annotations, and after using each model to re-annotate the code. They measured precision and recall of the model's annotations compared to the human-written annotations (another type reconstruction experiment), and reduction in the number of warnings issued by NullAway after each model's annotations were applied. NullGTN outperformed other models, achieving a recall of 89% and a precision of 60%, reducing the number of NullAway warnings by 60%.

*2.4.4 Summary of Bias in Prior Evaluations.* Both the WPI and NullGTN evaluations used type reconstruction experiments: given benchmarks that passed a checker, they assessed inference effectiveness based on how well the tool recovered human-written annotations. We shall show in section 3 that this methodology introduces a systematic bias in the experiments: when adopting a nullness checker, programmers not only add nullability annotations, but also modify parts of the code (e.g., adding null checks) to satisfy the checker. We show that these code modifications make the checking and inference problems easier, thereby biasing the experiment away from a realistic deployment scenario, where the target program has never been modified to support typechecking.

In the Nullaway Annotator evaluation, the authors did not perform type reconstruction experiments, avoiding the bias above. However, their approach solely assessed inference effectiveness by measuring the NullAway error reduction after inference. Implicitly, this methodology assumes that a larger NullAway error reduction indicates the inferred annotations are better, but their work does no evaluation of annotation quality to substantiate this assumption.

## 3 Analysis of Bias in Type Reconstruction Experiments

To investigate systematic bias in type reconstruction experiments, we performed a study on projects used in prior experiments. Our hypothesis was that using previously-checked code simplifies the checking and inference tasks versus never-checked code, the envisioned target for nullability inference tools. Specifically, we aimed to answer these research questions:

**RQ1** When developers verify their code using a nullability checker, do they make code changes beyond adding annotations? If so, what kinds of changes do they make?
**RQ2** Do the code changes, if they exist, improve the effectiveness of nullability checkers?
**RQ3** Do the code changes, if they exist, improve the effectiveness of nullability inference tools?

"Yes" answers to these questions indicate that using already-checked code to evaluate a nullability inference tool is not a good choice, since developers have already resolved some challenging issues that would still be present in unannotated codebases that have not passed nullness checkers. Note that the goal of this sub-study is not to compare the inference tools but to analyze the results of running different combinations of checkers and inference tools on previously-checked programs to investigate the systematic bias we believe exists in type reconstruction experiments.

### 3.1 Methodology

We selected benchmarks from prior type reconstruction experiments (section 3.1.1). We ran each inference tool (Nullaway Annotator, WPI, and NullGTN) and checker (CFNullness and NullAway) on the benchmarks using the procedure in section 3.1.2. We collected raw data (section 3.1.3) and calculated the evaluation metrics (section 3.1.4). Section 3.2 reports the results, section 3.3 discusses how the results show bias in prior evaluations.

**Table 1: Java benchmarks used in the bias illustration study, showing the original checker used to type-check each project ("O.C."), the total non-comment, non-blank lines of code ("LoC") in the post-check version, and error counts before inference for both pre-check ("PreV") and post-check ("PostV") versions of the benchmarks across CFNullness (CF) and NullAway (NW).**

| Benchmark | O.C. | LoC | CF PreV | CF PostV | NW PreV | NW PostV |
|---|---|---|---|---|---|---|
| Butterknife | CF | 2332 | 32 | 32 | 64 | 61 |
| Cache2k | CF | 2632 | 68 | 69 | 41 | 42 |
| | NW | | 70 | 68 | 41 | 41 |
| FloatingActionButtonSpeedDial | NW | 8407 | 50 | 50 | 29 | 29 |
| Jib | NW | 8351 | 241 | 237 | 60 | 56 |
| Meal-planner | NW | 1200 | 31 | 12 | 20 | 3 |
| Nameless | CF | 1934 | 27 | 27 | 21 | 21 |
| Picasso | NW | 8698 | 287 | 266 | 55 | 50 |
| Table-wrapper-api | CF | 1484 | 61 | 10 | 12 | 8 |
| Table-wrapper-csv-impl | CF | 950 | 16 | 25 | 4 | 6 |
| **Total LoC/Avg. Errors:** | | **35988** | **88.3** | **79.6** | **34.7** | **31.7** |

*3.1.1 Benchmark Selection.* Our benchmarks in this sub-study are Java projects that were manually annotated to satisfy NullAway or CFNullness. We considered 23 candidate benchmarks from prior type reconstruction experiments: 11 from the WPI study [Kellogg et al. 2023], and 12 from the NullGTN study [Siddiqui and Kellogg 2024]. We narrowed the set to projects for which we could identify commits clearly corresponding to before and after the nullness checkers were introduced. We searched for commits where nullability annotations were added, checker dependencies were introduced, and code changes were made to address nullability issues. We only included a commit if it contained specific evidence of its relevance to nullness checking (e.g., a commit message mentioning fixes for checker errors). Projects in which nullability checking was introduced over multiple small commits, interspersed with many unrelated changes, were excluded to avoid confounds. 13 of the 23 candidates were excluded based on these criteria. We also excluded one project (Caffeine) as it caused WPI to crash. For the Cache2k project, the developers initially used NullAway to verify their program, but later switched to CFNullness; we included each version as a separate benchmark.

For each benchmark, we chose commits corresponding to the pre- and post-check verion of the code. In most cases, we chose the commit immediately preceding the post-check code as the pre-check code. In cases where nullability changes were spread over multiple commits, we discussed and came to consensus on the best pre- and post-check versions, documenting our reasoning.

Our final list of 9 projects is provided in table 1, along with details on which checker was initially used to type check each project and the total non-comment, non-blank lines of code for the post-check version. Note that almost all of the selected benchmarks had some nullability annotations before being checked. We manually inspected the related build scripts and concluded that in most cases, these annotations were added for documentation purposes. However, the projects Table-wrapper-api and FloatingActionButtonSpeedDial had used FindBugs [Pugh et al. 2008] as a checker before, and Nameless had been checked by JetBrains' nullability tools [JetBrains 2006]. In keeping with our choice to include both versions of the Cache2k project, we still include these projects.

*3.1.2 Experimental Procedure.* To inspect the differences and code changes between pre- and post-check versions, we performed the following steps: 1) Count the nullness annotations and nullness-related `@SuppressWarnings` in both pre- and post-check versions. 2) Remove all such annotations from both versions, and also remove comments, empty lines, and all import statements. 3) Calculate the diff between the versions. 4) Manually categorize each nullability-related change.

Then, for each benchmark, we did the following to evaluate checking and inference effectiveness:

For the pre-check version: 1) Remove existing annotations, including `@SuppressWarnings` as it prevents checkers from analyzing certain parts of the code. 2) Run the checkers (CFNullness and NullAway). 3) Run inference tools Nullaway Annotator, WPI, and NullGTN. 4) Run the checkers on the annotated versions created by the tools in step 3.

For the post-check version: 1) Run the checkers. 2) Trim the existing annotations including `@SuppressWarnings`. 3) Run the checkers on the trimmed version. 4) Run the inference tools on the trimmed version. 5) Run the checkers on the versions created by the tools in step 4.

*3.1.3 Data Collection.* To compare the pre- and post-check versions, we counted the number of added annotations and the occurrences of each category of nullability-related code change in the post-check version. After each run of the checkers in the above steps (section 3.1.2), we logged the number of nullability errors emitted by the checker.

*3.1.4 Evaluation Metrics.* We used the number of errors emitted by the checkers as our main evaluation metric in this sub-study, as a useful proxy metric for the effectiveness of the checkers (further discussion in section 5.2.2). We use relative error reduction to compare configurations. For example, if CFNullness reports 15 errors on the pre-check version of a benchmark and 10 on the post-check version, the error reduction is $\left(\frac{15-10}{15}\right) \times 100 = 33.3\%$.

## 3.2 Results

*3.2.1 RQ1.* Table 2 shows our categorization of the differences between the pre- and post-check version of each benchmark. The data include how many `@SuppressWarnings`, `@Nullable`, and `@NonNull` annotations were added, and also the number of code changes in each category that we identified. The results show that code changes went beyond just adding nullability annotations. Many null checks were added (`C1`, e.g., `if (x != null) { ... }`) to guard dereferences. Calls to `Objects.requireNonNull` (`C2`) were used to fail fast when unexpected null values are encountered. Programmers also add field initialization (`C3`), mark fields as `final` (`C4`), modify parameters or types in method signatures (`C5`), use `this.X` instead of `X` in constructors (`C6`), define new methods or constructors (`C7`), adjust method arguments when calling methods (`C8`), change the return value when returning something while preserving the return type (`C9`), and changing the type of the fields (`C10`). Beyond the categorized code changes, many other changes (`Ot` in table 2) do not fit neatly into the categories (C1-C10). For example, in Meal-planner, field names were changed, and the data representation was updated, such as replacing standard collections with immutable data structures like `ImmutableMap`. In Butterknife, programmers modified the logic of control structures, including changes to `if` statements and `for` loop conditions. Our artifact describes each change.

These results show that programmers not only add annotations but also modify their code when integrating a checking tool. We hypothesized that they made these changes to resolve checker errors. To test this hypothesis, we next evaluated whether the code changes make checking simpler.

*3.2.2 RQ2.* To investigate whether these code changes contribute to more effective typechecking, we analyzed the error counts from the checkers on the pre- and post-check versions of the benchmarks (with nullability annotations trimmed) before applying any inference tools. Table 1 (four right-hand columns) shows that the number of errors in the pre-check version of most benchmarks is higher than in the post-check version, for both checkers. For example, in the Meal-planner benchmark, CFNullness issues 12 errors on the post-check version compared to 31 in the pre-check version, a 61% reduction. Similarly, NullAway issues 85% fewer errors on the post-check version of this benchmark. These significant reductions show that the non-annotation code changes contributed to resolving errors, even before applying inference tools.

**Table 2: Differences in post-check benchmarks. Added Annotations are: `@SuppressWarnings` (SW), `@Nullable` (Nul), and `@NonNull` (Non). Code Changes are: C1 (null check), C2 (`requireNonNull()`), C3 (initialize field), C4 (`final`), C5 (signature change), C6 (add `this`), C7 (new method/constructor), C8 (argument changed), C9 (return value changed), C10 (field type changed), and Ot (Others).**

| Benchmark | Added Annotations | | | Code Changes | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | SW | Nul | Non | C1 | C2 | C3 | C4 | C5 | C6 | C7 | C8 | C9 | C10 | Ot | Sum |
| Butterknife | 0 | 16 | 0 | 1 | 5 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 6 | 12 |
| Cache2k-CFNullness | 3 | 20 | 9 | 5 | 1 | 0 | 0 | 8 | 0 | 5 | 0 | 2 | 3 | 23 | 47 |
| Cache2k-NullAway | 3 | 105 | 8 | 4 | 0 | 0 | 0 | 1 | 0 | 9 | 2 | 1 | 0 | 11 | 28 |
| FloatingAction... | 0 | 23 | 0 | 7 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 7 |
| Jib | 0 | 52 | 0 | 19 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2 | 21 |
| Meal-planner | 0 | 1 | 0 | 0 | 0 | 18 | 9 | 0 | 17 | 0 | 0 | 2 | 3 | 5 | 54 |
| Nameless | 0 | 1 | 11 | 0 | 0 | 0 | 17 | 0 | 0 | 0 | 0 | 0 | 0 | 3 | 20 |
| Picasso | 2 | 51 | 34 | 39 | 0 | 2 | 2 | 5 | 0 | 1 | 5 | 0 | 0 | 10 | 64 |
| Table-wrapper-api | 17 | 13 | 0 | 3 | 6 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 10 |
| Table-wrapper-csv-impl | 1 | 17 | 0 | 3 | 1 | 3 | 3 | 3 | 0 | 5 | 0 | 1 | 0 | 4 | 23 |
| **Total** | 26 | 299 | 62 | 81 | 13 | 23 | 31 | 17 | 17 | 20 | 7 | 6 | 6 | 65 | 286 |

Table-wrapper-csv-impl is an outlier, with more errors in the post-check version. This increase is caused by new code in the post-check version that seems not to have been null-checked, since it causes several new errors. We speculate it was being edited independently of the annotation process.

*3.2.3 RQ3.* To investigate whether these code changes contribute to more effective inference, we analyzed whether applying inference on the pre- or post-check versions led to fewer errors. Table 3 shows that for most benchmarks, across all combinations of inference tools—WPI, NullGTN (NGT), and Annotator (ANN)—and checkers, inference reduces errors more on the post-check code. For example, when running CFNullness with WPI's inferred annotations (WPI + CF) on the pre-check and post-check versions of Table-wrapper-api, the number of errors decreased from 12 to 5 (a 58% reduction). Similarly, running NullAway on the inferred version produced by Nullaway Annotator (ANN + NW) on this benchmark resulted in a 100% error reduction (from 12 to 0).

These results suggest that type reconstruction experiments overestimate the effectiveness of inference tools: the type reconstruction task for the "post" version is easier than the task of annotating the same codebase before it has ever been annotated by a human (the "pre" version).

*3.2.4 Example.* To illustrate how code changes beyond adding annotations can help both checkers and inference tools, consider this change in the `Dispatcher.java` file from Picasso:

```
+ if (connManager != null) {
    networkInfo = connManager.getActiveNetworkInfo();
+ }
```

CFNullness raises an error here on the pre-check version before the above change, without using any inference tool. However, in the post-check version, the addition of the null check ensures that `connManager` is non-null within the `if` body, allowing CFNullness to verify that it is safe to dereference it—preventing the error. Moreover, because CFNullness now knows that `connManager` is non-null, an inference tool like WPI that uses CFNullness as a subroutine will no longer propagate nullability because of how `connManager` is used—improving the inference result, too.

## 3.3 Discussion

We have shown that type reconstruction experiments on previously-checked benchmarks introduce a systematic bias: they overestimate the effectiveness of the studied nullability inference tools. Developers change their code as they annotate in a way that makes both typechecking and type inference easier, so these experiments do not evaluate the tools on their ability to handle the complexities of unchanged, unchecked code that a programmer has not modified. We suggest using never-checked benchmarks instead of previously-checked ones for more accurate evaluations.

**Table 3: Error counts after inference for the pre- and post-check versions of the benchmarks, using WPI, NullGTN (NGT), and Annotator (ANN) inference tools with both CFNullness and NullAway. The average counts are rounded to the nearest whole number.**

| | WPI + CF | | WPI + NW | | ANN + CF | | ANN + NW | | NGT + CF | | NGT + NW | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Benchmark** | **Pre** | **Post** | **Pre** | **Post** | **Pre** | **Post** | **Pre** | **Post** | **Pre** | **Post** | **Pre** | **Post** |
| Butterknife | 31 | 30 | 57 | 52 | 27 | 25 | 39 | 34 | 41 | 41 | 71 | 67 |
| Cache2k-CFNullness | 21 | 20 | 7 | 11 | 29 | 5 | 1 | 1 | 51 | 72 | 117 | 121 |
| Cache2k-NullAway | 23 | 21 | 7 | 7 | 31 | 26 | 2 | 1 | 109 | 107 | 118 | 117 |
| FloatingAction... | 48 | 48 | 28 | 28 | 34 | 35 | 3 | 0 | 53 | 53 | 30 | 28 |
| Jib | 462 | 440 | 103 | 88 | 264 | 253 | 12 | 2 | 301 | 179 | 87 | 69 |
| Meal-planner | 20 | 7 | 20 | 2 | 22 | 10 | 4 | 1 | 299 | 278 | 24 | 6 |
| Nameless | 3 | 3 | 4 | 4 | 26 | 26 | 0 | 0 | 35 | 15 | 21 | 21 |
| Picasso | 623 | 612 | 128 | 122 | 215 | 193 | 14 | 2 | 26 | 26 | 108 | 98 |
| Table-wrapper-api | 12 | 5 | 12 | 0 | 10 | 24 | 3 | 1 | 374 | 345 | 30 | 27 |
| Table-wrapper-csv-impl | 8 | 6 | 6 | 3 | 16 | 34 | 2 | 0 | 47 | 41 | 3 | 4 |
| AVERAGE | 125 | 119 | 37 | 32 | 67 | 63 | 9 | 4 | 134 | 126 | 61 | 56 |

*3.3.1 Other Type Inference Contexts.* Our results raise the question of whether bias in type reconstruction experiments is nullability-specific, or whether it might apply to type reconstruction experiments in other contexts. Type reconstruction experiments are the standard way to evaluate type inference for modern optional type systems, such as MyPy or TypeScript; e.g., the standard benchmark for machine-learning based Python type inference tools [Li et al. 2023; Mir et al. 2022; Peng et al. 2022, 2023] is ManyTypes4Py [Mir et al. 2021], a type reconstruction benchmark.

There are important differences between inferring nullability annotations and inferring types for dynamically-typed languages like Python. Python type inference focuses on the "rare type" problem (i.e., there is a long tail of user-defined types, and the hardest part of inference is handling that long tail); by contrast, nullability annotation inference is more of a classification problem. There are also significant differences between the relevant checkers, which might influence the behavior of developers: nullability checkers are typically relatively opinionated compared to gradual typecheckers about what code patterns they will verify, and nullness checkers also typically have higher false positive rates. Both of these tendencies may encourage developers to make more changes to their programs to help a nullability checking tool, making inference easier. Nonetheless, our investigation raises an important question with respect to the validity of the experimental designs used by recent work in gradual type inference, and future work should investigate whether those tools perform as well on fully-unannotated benchmarks as in type reconstruction experiments.

## 3.4 Threats to Validity

There are threats to this study's external validity: we only studied nine projects, since few human-annotated benchmarks exist, and the projects that we did study may not be representative. The effects we observed were consistent across the different projects, checkers, and inference tools, which gives us confidence against these threats. Some of our analysis was conducted by hand, and we could have made mistakes. We mitigate this threat by releasing records of our decisions and judgments, our code, and our data. Several projects had been checked by some nullability detection tool before the "pre-check" version. This confound means that our experiments may underestimate the extent of changes induced by adopting a checking tool.

## 4 Judging Annotation Quality Without Ground Truth

Section 3 showed that type reconstruction experiments overestimate a nullability inference tool's effectiveness compared to how it will be deployed. This section explores how to judge the quality of inferred annotations without ground truth. We first describe an ideal experiment (section 4.1) that avoids the bias from section 3; however, it is clearly infeasible for cost reasons. Instead, we turn to a theoretical analysis of the properties of the "best" annotations that an "ideal" inference tool would infer (section 4.2), as well as a discussion of alternative definitions of "best" and why we discarded them (section 4.3). The theoretical analysis in this section provides the basis for the metrics used in our evaluation of inference tools on unannotated benchmarks (section 5.2).

### 4.1 Ideal Annotation Reconstruction Experiment

Ideally, to evaluate a nullability inference tool, we would first select a large, diverse set of benchmarks that have never been annotated or checked for nullability. We would then hire developers to annotate these benchmarks for nullability. These developers would not be allowed to modify the code, except to suppress warnings in cases that cannot be resolved by annotations alone, similar to the intended functionality of the inference tools under investigation. This would provide an accurate and comprehensive set of ground-truth annotations to compare against the output of the inference tools, and would exclude code changes like those that biased prior type reconstruction experiments. However, this experiment would be prohibitively expensive, as annotating a non-trivial codebase can take days or weeks. Asking humans to do so under the artificial restriction of not changing code for an experiment is impractical.

### 4.2 Defining the "Best" Annotations

For a program $P$ that has never previously been annotated for nullness, how can we measure the quality of an inference tool's output? Given a definition of the "best" annotations for $P$, one approach would be to measure the difference between the tool's inferred annotations and the "best" annotations. A drawback of this approach is that it may not accurately capture how a developer would annotate $P$; as shown in section 3.2.1, when manually annotating a program, developers almost always modify the code in the process, e.g., to fix bugs. Since none of the inference tools we consider perform such code changes, we put this issue aside in this work, and proceed with studying how to best annotate a subject program without any code changes. Incorporating the impacts of code changes is an important topic for future work; a future inference tool that both infers annotations and proposes typical code changes could possibly outperform extant tools.

Defining the "best" annotations for a program based on whether variables may or may not be `null` is intuitive, but flawed. We are primarily concerned with annotations on the types of fields, formal parameters, and method returns, since both nullability-checking tools that we study do type inference for local variables. Intuitively, one might think that a `@Nullable` annotation should appear on any such type if the corresponding variable (or returned value) might be assigned `null` at run time. Unfortunately, this definition does not work for Java fields, all of which are initially set to `null` according to the language semantics [Gosling et al. 2021]. The simple definition would require making all fields `@Nullable`, an unsatisfactory solution.

Instead, we informally define the "best" nullness annotations for $P$ as follows (a corresponding formal definition appears later in this section). We assume that $P$ is a "closed" program, i.e., there is no unknown code that may invoke methods in $P$; we defer discussion of libraries, where this does not hold, to later in the section. Then, the type of a field or formal parameter in $P$ should be `@Nullable` if *there exists some read* of the field or formal that may observe the value `null`. Similarly, a return type should be `@Nullable` if there exists some read of the return value at a call site that can observe `null`. If such a location is read at least once, and it is always initialized to a non-`null` value before it is read, it should *not* be marked as `@Nullable`.

Note that the definition accommodates *any* protocol for field initialization that ensures fields will always be initialized before use; any such field need not be marked as `@Nullable`. Field initialization schemes in real-world code may be complex [Summers and Müller 2011], and we require a definition that allows for this complexity. Our definition accommodates both the lazy initialization of fig. 1 from section 2 and the lifecycle-based initialization of fig. 2.

A drawback of this definition is its reliance on possible run-time behaviors of a program. Since it only constrains annotations on locations that are read at least once, it has nothing to say about how annotations should be placed in dead code. For our purposes, dead code includes code that never executes, and also formal parameters, function return values (for all call sites), and fields that are never used. Programs may contain dead code for any number of valid reasons (e.g., code for a new feature that is partially implemented and not yet enabled). Checking tools are typically run on all source code in a program, so in adopting such a tool, dead code would also need to be annotated. Nevertheless, we adopt a definition tied to run-time behaviors to avoid tying it to specific static reasoning techniques for nullness, which could make the definition too tied to a particular static checker. Also, leveraging run-time semantics is appropriate in this case since nullness checking is not built into the Java language semantics; a Java program may still be executed even if it has type errors from a nullness checker, so its run-time behavior is still well-defined.

Automatically computing the best annotations for a program according to the above definition is undecidable (since even determining if code may execute is undecidable [Rice 1953]). In our evaluation, the definition serves as a guide when manually evaluating the `@Nullable` annotations inserted by different inference engines (section 5.2.1). Regarding dead code, in our manual evaluation we did not attempt to determine if code was truly reachable from an entrypoint. Instead, we did a local inspection to discover calls to methods to learn how they are expected to behave.

*Formalization.* To make our definition of the "best" annotations more rigorous, we formalize the concept in terms of program execution and memory interactions. We base this formalization on the Java memory model as described in *The Java$^{TM}$ Language Specification* (JLS) Section 17.4 [Oracle 2015], which defines the possible behaviors of a program.

The following definitions establish a framework for reasoning about nullness based on program traces, read actions, and dereference chains. This framework allows us to rigorously state conditions under which an element should be annotated as `@Nullable` and provides a formal basis for ensuring null safety.

*Definition 4.1.* [Action] An **action** $a$ is described by a tuple $\langle k, v, u \rangle$, where $k$ is the kind of action (e.g., read, write, lock, unlock, etc.), $v$ is the variable or monitor involved in the action, and $u$ is the value associated with the action.[1]

*Definition 4.2.* [Program Trace] A **program trace**, denoted as $T$, is a global sequence of actions $\langle a_1, a_2, \dots, a_n \rangle$ that represents the execution of a program during a single run.

*Definition 4.3.* [Read Action] A **read action** occurs when a variable $v$ is accessed, and a value $u$ is transmitted from the main memory to the working memory of the thread executing the program.

*Definition 4.4.* [Best Annotation Rule] The type of the program element $v$ at its declaration should be annotated as `@Nullable` if:

$$\exists\, a \in T,\ \text{Kind}(a) = \text{Read} \land \text{Variable}(a) = v \land \text{Value}(a) = \text{null},$$

[1]Note: This definition adapts the action formalism from JLS 17.4.2 [Oracle 2015] and omits the thread $t$ component, as threads are not directly relevant to this work.

where $\text{Kind}(a)$ denotes the kind of action $a$ (e.g., read, write, lock, etc.), $\text{Variable}(a)$ represents the program element $v$ involved in action $a$, and $\text{Value}(a)$ is the value associated with the action $a$.

*Definition 4.5.* [Dereference Chain] The **dereference chain** dereferences() is a partial order over actions in the program trace $T$ [Oracle 2015]. Formally, for actions $r$ and $a$, $\text{dereferences}(r, a)$ holds if and only if $r$ is a read action that observes the reference of an object $o$, and $a$ is a subsequent action that accesses $o$, such as reading or writing a field (e.g., $o.field$), invoking a method (e.g., $o.method()$), or accessing an array element (e.g., $o[index]$). This partial order reflects the causal relationship between $r$ and $a$, where $r$ provides the reference required by $a$.

*Definition 4.6.* [Dereference(x)] For a program element $x$, a **dereference(x)** is the set of all actions $a$ in the program trace $T$ that access $x$. Formally:

$$\text{Dereference}(x) = \{a \in T \mid \exists r \in T,\ \text{dereferences}(r, a) \land v(a) = x\},$$

*Definition 4.7.* [Null Safety] A program $P$ is **null-safe** if for every program element $x$ and for every action $a \in \text{Dereference}(x)$, all read actions $r$ in the dereference chain for $a$ provide a non-`null` reference. Formally:

$$\forall x \in E_P,\ \forall a \in \text{Dereference}(x),\ \forall r \in T,\ \text{dereferences}(r, a) \implies \text{Value}(r) \neq \text{null},$$

where:

- $E_P$: The set of all program elements in the program $P$, including variables, fields, formal parameters, and method return values.

*Proof Sketch for Null Safety.* We aim to show that if a program is annotated according to the *Best Annotation Rule* (Definition 4.4) and a sound nullability type checker reports no errors, then the program satisfies the definition of null safety (Definition 4.7).

Assuming the program is analyzed by a sound[2] nullability type checker, the proof proceeds by case analysis on the dereference chain for a program element $x$:

- **Case 1:** At least one read action $r$ in the dereference chain retrieves a `null` reference. Since $\exists r \in T$ where $\text{Value}(r) = \text{null}$, the *Best Annotation Rule* (Definition 4.4) requires that $x$ be annotated as `@Nullable`. A sound nullability checker prevents dereferencing such elements without a null check; if unguarded, it reports an error, ensuring the program does not execute with unsafe dereferences.
- **Case 2:** All read actions $r$ in the dereference chain retrieve non-`null` references. If $\forall r \in T,\ \text{dereferences}(r, a) \implies \text{Value}(r) \neq \text{null}$, then $x$ is not annotated as `@Nullable`. By Definition 4.7, null safety holds since no dereference chain retrieves a `null` value.

*Libraries.* The above definition assumes that a program $P$ is "closed" and contains all relevant code that may invoke any method defined in $P$. However, this assumption does not cover libraries, which are "open" programs with public methods that may be invoked by unknown client code. In fact, libraries themselves may not contain any invocations of some of their public API methods (excluding test code). A question remains of how to best annotate such API methods. For these methods, we follow the alternative principle of "accurate API documentation" (section 4.3.3) by requiring that a parameter of a public API should not be `@Nullable` unless passing `null` can lead to an NPE within the library. Beyond that, in the evaluation we make a best effort to determine whether such a parameter should be `@Nullable` using both code and any available documentation.

[2]A "sound" nullability checker is one that rejects all programs with unsafe dereferences. In practice, extant nullability checking tools are at best "sound-y" [Livshits et al. 2015]: that is, they are sound with some caveats (e.g., sound except in the presence of integer overflow, reflection, etc.). This proof illustrates that our definition is theoretically sensible.

### 4.3 Discussion: Alternative Principles for "Best" Annotations

This section discusses alternative principles for the "best" annotations that we considered when devising the definition of section 4.2. These principles are all "good" in a platonic, abstract way: ideally, a nullness annotation inference tool could achieve all of them at once. In practice, however, there are trade-offs between them. We discuss each principle in turn and discuss the scenarios for which each principle is important, their trade-offs, and why we prefer the definition in section 4.2.

*4.3.1 Accuracy with respect to run-time semantics.* It is desirable for the set of annotations to be *correct* in the sense of soundness (every program element that may be null is marked `@Nullable`) and precision (only program elements that may be null are marked `@Nullable`). However, soundly and precisely determining whether elements may be null at run time is undecidable [Rice 1953], and hence fully sound, precise, and terminating inference cannot be achieved. Further, as noted in section 4.2, full accuracy does not yield useful results for Java fields, which are all initially assigned `null`.

*4.3.2 Error reporting locations.* It is desirable for the set of annotations produced by an inference tool to lead a nullness checker to produce warnings that are "close" to the root causes of those warnings. For a true null pointer bug in the program, ideally the warning would appear near where the developer would prefer to fix the bug. And, for code that is correct but beyond the verification capabilities of the checker, ideally the warning would appear near the unverifiable code pattern. The fundamental downside of defining "best" annotations based on the location of warning reports is that it relies on knowing how developers would choose to address warnings. In practice, it is not feasible to collect this information from developers at scale, and preferences may vary across developers or teams. Further, this principle does not help us judge the quality of annotations that are not involved in producing a checker warning.

*4.3.3 Accurate API documentation.* It is desirable for the "best" annotations to accurately represent the entry- and exit-points of the program with respect to their nullability, while leaving future maintainers implementation freedom where possible. Consider the case of a library that is intended to be used by unknown client code. For this case, it may be desirable to infer more `@Nullable` annotations that what are strictly required to allow a checker to verify the library. E.g., for the fig. 3 example from section 2, the inferred `@Nullable` annotations on the `sumLengths` parameters are not required to verify its safety, but they accurately capture its ability to accept `null` arguments.

One could define the "best" annotations for libraries based on all possible invocations of a public API method: require `@Nullable` on any parameter where passing in `null` cannot cause an NPE inside the library. But, this definition ignores developer intent, which may be evident in non-code artifacts like API documentation. We want a methodology that accommodates tools that can reason about such non-code artifacts, like ML-based approaches. So, we only require that a parameter should *not* be `@Nullable` if passing in `null` can lead to an NPE within the library. This principle is useful for evaluating annotations on public APIs, but it does not give any guidance for closed programs, which is why we do not make it the core principle of our best annotations definition.

## 5 Comparative Evaluation Without Type Reconstruction

Our goal is to conduct a comprehensive evaluation of the three extant nullability inference tools that avoids the bias of type reconstruction experiments identified in section 3. To that end, we formulate these research questions (RQs):

**RQ4** What is a standardized evaluation methodology to fairly evaluate the effectiveness of nullability inference tools that avoids the bias inherent in a type reconstruction experiment?

**RQ5** What are the comparative results of the different nullability inference tools when evaluated using the proposed standardized evaluation methodology?

## 5.1 Benchmarks

We chose the Normalized Java Resource (NJR-1) dataset [Utture et al. 2020], which consists of 293 real-world Java programs collected from GitHub. This dataset was selected because its creators had already performed the challenging task of normalizing the programs, making it easier to run various checking and inference tools. Additionally, it has been used in multiple studies [Kalhauge and Palsberg 2019; Le-Cong et al. 2022; Mir et al. 2024; Utture et al. 2022; Utture and Palsberg 2023]. After excluding benchmarks incompatible with Java 11, our final selection included 255 programs, totaling 1.44 million lines of code (LoC). Its programs have an average of 5.6K lines, a maximum of 52K lines, and a minimum of 440 lines, excluding comments and blank lines.

## 5.2 Proxies for the Idealized Inference Tool

Defining the "best" nullability annotations for a given program is more complex than it initially appears (see section 4 for discussion). While aligning annotations with run-time behavior is a good starting point, it does not fully account for the intricacies introduced by initialization patterns and the specific behaviors of different null-checking tools. So, we answer **RQ4** by proposing a set of proxies—manual analysis of annotation quality, remaining error counts after inference, and using dynamically-measured nullability as ground truth—that approximate an idealized inference tool: that is, each proxy cannot solely determine the "best" annotations, but together they provide a holistic view of the quality of the annotations produced by the nullability inference tools.

*5.2.1 Annotation Quality.* For our manual assessment, we sampled 150 annotations that were added by only one of the tools and 150 annotations that were missed by only one tool: that is, we sampled places where one tool disagrees with the others. This approach focused on understanding the differences between the three tools, informing potential improvements and further research. We did not analyze cases where all tools produced the same annotations, as such cases do not illuminate the differences we aimed to explore. Two authors evaluated the quality of these annotations and recorded their assessments. Initially, they disagreed on 10 cases, but after discussion and review, all disagreements were resolved, leading to a consensus on every case. The Cohen's Kappa [Cohen 1960] value for their initial independent evaluations was 0.93, indicating near-perfect agreement. This study highlights the mistakes and good decisions made *uniquely* by each inference tool. We manually judged each case guided by our definition of "best" annotations in section 4.2, focusing on whether the annotation reflects the program's intent and whether we would keep the annotation in the code.

Figure 4 shows our results as a Venn diagram. There were 71 cases where only Annotator was correct, 27 cases for WPI, and 22 cases for NullGTN. There were 29 cases where only Annotator was wrong, 73 cases for WPI, and 78 cases for NullGTN. A split $(+x/-y)$ is shown for each number, where $x$ corresponds to inferring `@Nullable` and $y$ to *not* inferring `@Nullable`. We discuss results for each tool in turn.

*Annotator.* For the 43 cases where only Annotator correctly inferred `@Nullable`, in 18 cases the disagreement stemmed (sometimes indirectly) from WPI not marking a field as `@Nullable`, due to a bug in WPI's handling of uninitialized fields in constructors (we reported all discovered bugs to the maintainers). 9 cases were due to WPI's lack of support for generic types. In the remaining 16 cases, there was no clear pattern. The 28 cases where only Annotator correctly avoids inferring `@Nullable` all corresponded to field initialization. 8 cases were due to better analysis of initialization logic in NullAway, and 20 cases were due to Annotator's avoidance of `@Nullable` annotations that increase the final error count (section 2.2). Figure 2 is an example of the latter: Annotator's error reduction heuristic correctly led it to not annotate the `name` field as `@Nullable`.

For the 7 cases where only Annotator wrongly inferred `@Nullable`, the main cause was imprecision in NullAway's dataflow analysis. The 22 cases where only Annotator wrongly missed inferring `@Nullable` were due to its error reduction heuristic, which in these cases led it to miss a good `@Nullable` annotation.

*WPI.* For the 18 cases where only WPI correctly inferred `@Nullable`, 15 were again due to Annotator's error reduction heuristic causing it to miss a good annotation. The remaining 3 cases were due to CFNullness having better models of Java library methods like `Method.invoke` and `Map` methods. For the 9 cases where only WPI correctly avoided inferring `@Nullable`, there was no clear pattern. The 32 cases where only WPI wrongly inferred `@Nullable` were all variants of field initialization issues. 16 cases involved initializer methods invoked elsewhere, like `onStart` in fig. 2. In 9 cases fields were initialized in methods called by the constructor, which is handled by NullAway but not CFNullness. There were 3 cases of protected fields initialized in subclasses, and 4 cases of static fields initialized in `main`. Finally, for the 41 cases where only WPI wrongly missed inferring `@Nullable`, 13 cases were due to the bug in WPI's field initialization handling mentioned earlier, 18 cases were due to lack of support for generics, and there was no clear pattern in the remaining 10 cases.

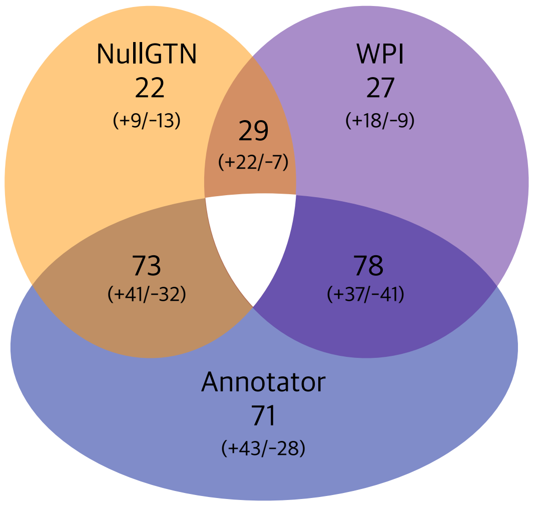

**Fig. 4: Venn diagram illustrating number of cases where each tool handled a disagreement correctly.**

*NullGTN.* There were 9 cases where only NullGTN correctly inferred `@Nullable`, and they were all on unused method parameters. We treated these annotations as correct since they safely provide additional flexibility to callers to pass in `null`. Surprisingly, in this study we found no cases like that of fig. 3 where only NullGTN correctly made a *used* parameter of a public API `@Nullable`. In an additional manual inspection of 50 uncalled public API methods, we only found one case like fig. 3 where NullGTN added the desired annotation, but found many cases where it added erroneous annotations on primitive types (more discussion below). For the 13 cases where only NullGTN correctly refrained from inferring `@Nullable`, 4 were cases of complex field initialization. In the other 9 cases, Annotator and WPI incorrectly made a parameter `@Nullable` due to imprecise analysis of callers, which in fact can never pass `null`.

For the 41 cases where only NullGTN wrongly inferred `@Nullable`, 21 were due to NullGTN incorrectly marking primitive types as `@Nullable`, which the maintainers have acknowledged is a bug. Nonsensical inferences like these may stem from NullGTN's use of ML for type inference; deductive approaches easily avoid these mistakes. The remaining cases had no clear pattern and were difficult to understand. For example, there were five cases where the event object parameter passed to a GUI event callback were marked as `@Nullable`, which makes little sense. There was no clear evidence of why NullGTN incorrectly inferred `@Nullable` for these cases—a result of the inscrutability of deep learning techniques. For the 37 cases where NullGTN wrongly missed inferring `@Nullable`, there was clear evidence of nullability, such as null literal assignments, null checks, and null returns from methods; the inscrutability of deep learning means we cannot explain why NullGTN did not infer `@Nullable` for these cases.

*5.2.2 Remaining Error Count.* From an automation perspective, the "best" annotations could be those that require the fewest code changes and the least effort from the programmer to satisfy a nullness checker. In this sense, how many errors remain after applying the tool's inferred annotations is a proxy for the quality of those annotations, particularly in how well they align with the program's run-time behavior.

Clearly having no errors remain is desirable, since then the program has been verified. When some errors do remain (nearly all cases in practice), absolutely minimizing the number of errors (as done by Annotator) is not always beneficial, as discussed in section 2.2. Section 5.2.1 showed that more often than not, Annotator's error minimization heuristic did lead to better overall annotations compared to WPI. Here, we measure remaining error count as another complementary proxy for annotation quality, given the intuitive benefits of a smaller number of errors and the evidence from section 5.2.1 that with fewer errors the annotations may be of higher quality.

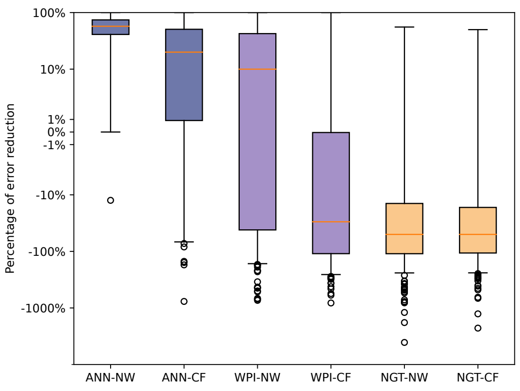

**Fig. 5: Box plots depicting the distribution of percentage error reduction reported by two nullness checkers across all three inference tools. ANN:Annotator, NGT:NullGTN, WPI:WPI. NW: NullAway, CF: CFNullness.**

For this experiment, after applying the annotations inferred by each tool, we ran both NullAway and CFNullness on the annotated code to collect remaining errors. The distribution of percentage error reduction for each combination of inference and checking tool appears in the box plots of fig. 5, where 100% indicates complete error removal and negative values signify an increase in errors.

*Annotator.* Annotator decreased the number of errors reported by NullAway by an average of 50% (-12.5% to 100%). Annotator was less effective in reducing errors reported by CFNullness, with an average reduction of 25% (-766% to 100%). This difference is expected, as Annotator is designed to minimize NullAway errors, and also CFNullness does stricter checking than NullAway.

*WPI.* On average, WPI increases the final error count from the checkers, with a -21% reduction with NullAway and a -68% reduction with CFNullness. It reduces the final error count in 134 cases with NullAway and 62 cases with CFNullness. WPI is not designed to minimize errors, which explains the low error reduction rates. Instead, WPI is intended to align with CFNullness—a conservative checker that errs on the side of warning about possible null dereferences, leading to more errors.

*NullGTN.* NullGTN increased the final error count in most cases: it only achieved an error reduction in 24 cases for NullAway and 11 for CFNullness. Its average reductions were -118% for NullAway and -93% for CFNullness. NullGTN is not designed to minimize errors from any tool and is trained on code that was not run through checking tools (section 2.3), which could explain the very low error reduction rates.

*5.2.3 Dynamic Nullability.* Another proxy for the ground-truth is *dynamic nullability*—whether a program element is null or non-null at run time. This proxy is necessarily limited: like any dynamic analysis, it depends on the quality of the inputs to the analyzed programs. This approach also does not directly evaluate the quality of the inferred annotations; our section 4.2 definition does *not* require that all locations that may be null at run time be marked `@Nullable` to account for field initialization. Nevertheless, dynamic nullability offers another useful view of the outputs of the inference tools.

Each benchmark in NJR contains a single main class, and NJR guarantees that running the main class will invoke at least 100 methods within the program. Of our 255 programs from NJR, we successfully collected data for 110 programs where the execution was successful and the Daikon invariant detection tool [Ernst et al. 2007] was able to generate dynamic traces. The remaining programs failed due to issues such as missing dependencies, missing resources, or problems with network or database access. Out of the 110 programs, only 33 programs had at least one null element identified by dynamic analysis; we focus our analysis on these programs. We collected dynamic traces for these programs using Daikon. We parsed Daikon's output and recorded which parameters, return values, and fields contain `null` at any point during program execution. We compared the result to the annotations inferred by our three inference tools.

We define the Missed Annotation Rate (MAR) as our primary metric:

$$\text{MAR}(\%) = \left(\frac{\text{Non-annotated nullable elements}}{\text{All nullable elements}}\right) \times 100$$

This metric measures the proportion of nullable elements identified by dynamic analysis that the inference tool failed to annotate. Lower MAR is usually better.

Out of 312 program elements in the 33 subject programs that we observed to be dynamically nullable, WPI marked 309 as `@Nullable` (MAR of 0.97%), Annotator marked 295 as `@Nullable` (MAR of 5.45%), and NullGTN marked 231 as `@Nullable` (MAR of 25.96%). WPI's low MAR is expected because it is based on the conservative CFNullness checker, which has a strong soundness guarantee. All three elements that WPI missed were due its lack of handling for generic types, which the WPI authors admit as a weakness (Section VI.E.1.b of [Kellogg et al. 2023]). Annotator can miss a `@Nullable` annotation for two reasons: either NullAway unsoundly does not detect the nullable flow into the location, or adding the `@Nullable` annotation increases the number of errors from NullAway. Out the 17 misses, 11 of them were due to the first reason, namely a call to a third-party routine that NullAway assumes does not return null. 6 of them were due to the second reason of leading to an increased NullAway error count. All these cases should have been marked `@Nullable` according to our section 4.2 definition. NullGTN annotates many fewer dynamically-nullable locations than the deductive approaches, but because its deep-learning-based model is not explainable, we cannot interrogate why.

Still, overall all three tools caught most of the null elements identified by dynamic analysis, with WPI being the most successful. The experiment shows a significant difference between Annotator and WPI: WPI tries its best to encode the program's *current* semantics, while Annotator uses NullAway as a crude fitness function for the *intended* semantics. A promising avenue for future work is better fitness functions for the intended semantics that better capture programmer intent.

## 5.3 Summary and Discussion

Our overall answer to **RQ5** is that while Nullaway Annotator stands out—it makes the fewest mistakes according to our manual analysis, and is the most effective at error reduction—all three tools have distinct strengths, and no one tool strictly dominates the others. Annotator's relative strength seems to stem from its design choice to optimize for minimizing errors, which automatically leads to leveraging contextual usage patterns in how program elements are used—for example, Annotator can avoid falling into the trap of annotating a field as `@Nullable` due to a buggy nullable dataflow in, if that field is dereferenced without null checks many times. WPI, in contrast, strictly adheres to the analysis provided by CFNullness, ensuring consistency but often over-producing `@Nullable` annotations due to the checker's conservative approach, particularly when dealing with complex object initialization protocols. WPI's reliance on a sound checker makes it effective in capturing run-time semantics, as shown by dynamic nullability, but less flexible in adapting to programmer intent compared to Annotator. NullGTN uses artificial intelligence to infer annotations based on patterns learned from open-source projects, making it capable of analyzing public APIs even when they are not called in the code. However, this method can lead to over-generalizations, such as marking primitive types as nullable, and struggles with poor code quality.

## 5.4 Threats to Validity

There are generalizability threats to our conclusions: we only evaluated 3 inference tools, only for Java, and only for nullability. What makes a high-quality inference tool for other languages or other analysis domains besides nullability may differ. While the NJR benchmarks are diverse, open-source projects, they may not be representative of e.g., industry code (which may be higher quality, better tested, etc.). Our analysis of annotation quality (section 5.2.1) was conducted by hand, and we may have made errors; we mitigated this threat by both having a second author double-check all judgments and by releasing our data and judgment records publicly. The definition of "best annotations" that guided our manual analysis was also chosen based on a theoretical analysis, and may not match what some developers would prefer. Our dynamic nullability experiment (section 5.2.3) was conducted on only the small group of NJR benchmarks that 1) could execute, and 2) had at least one null element detected by Daikon; both of these limit its generalizability. Moreover, the test cases provided with NJR may not be high quality: "cover at least 100 methods" is an arbitrary and not necessarily high bar. We mitigate these threats by not overly relying on the dynamic nullability results: it is one among several proxies. Finally, our two proxies fundamentally approximate the ground truth, and even together they might not capture some important aspect of nullability inference. We mitigated this threat with our manual annotation quality experiment.

## 6 Related Work

*Nullability Annotation Inference Tools.* The papers that introduced the three nullability annotation inference tools [Karimipour et al. 2023; Kellogg et al. 2023; Siddiqui and Kellogg 2024] are the most closely-related works, because each evaluated its tool in isolation. WPI and NullGTN were evaluated with type reconstruction experiments, which we showed in section 3 are a flawed way to evaluate these tools: they overestimate how well an inference tool would do in its intended deployment context. Because type reconstruction experiments cannot be conducted unless a developer has annotated the code, the subject programs in these experiments are artificially easy to annotate. The Nullaway Annotator paper [Karimipour et al. 2023] did not do a type reconstruction experiment, but instead evaluated the tool exclusively on its ability to reduce NullAway warnings. While this is a useful proxy, our experiments in section 5 show that it does not tell the whole story. We discuss prior experiments in more detail in section 2.4. Our work both corrects methodological flaws in prior work and evaluates all three tools fairly, on the same set of benchmarks, for the first time.

The Daikon invariant detector [Ernst et al. 2007] includes a mode to insert `@Nullable` annotations into the source code for any program element that it detects at run time contains a null value [The Daikon developers 2024]. This mode is one of our proxies in our comparative evaluation (section 5.2.3), because as a dynamic technique it is inherently limited by the quality of the available inputs for the target program—unlike the static techniques that we did evaluate. In a realistic deployment scenario, we do not believe that it is reasonable to assume that the target program will have a representative test suite available, so we do not consider dynamic inference tools. Another dynamic inference technique was proposed by Dietrich et al. [2023] based on using test cases to detect locations that might contain null at run time. It was evaluated with a type reconstruction experiment, and so its evaluation suffers from the same problems that we noted in the static tools' evaluations in section 3.

*Comparative Evaluations of ML-based Type Inference Tools.* The NullGTN paper [Siddiqui and Kellogg 2024] cites recent advances in machine-learning-based type inference for dynamically-typed languages like Python as an influence on their experimental design. These works use type reconstruction experiments. For example, the ManyType4Py benchmark [Mir et al. 2021] is the standard benchmark for Python, used in most recent works [Li et al. 2023; Mir et al. 2022; Peng et al. 2022, 2023]; this benchmark's task is type reconstruction. We discuss whether the issues we discovered with these experiments apply to this line of work in section 3.3.1.

*Comparative Evaluations of Deductive Type Inference Tools.* Classical type inference [Pierce 2002, Chapter 22] focuses on finding a complete typing for a program in a language's type system. Languages with global type inference often admit inference algorithms that are guaranteed to compute the principal (best) types, if they exist; in such cases, there is no need to compare the output quality of different inference algorithms. We focus on cases where we want a partial inference solution for a program that may not pass a checker (e.g., due to program bugs), so classical inference techniques do not directly apply. Some techniques to compute better explanations of type inference errors use a constraint-based approach [Loncaric et al. 2016; Pavlinovic et al. 2014; Zhang et al. 2017], aiming to satisfy as many of the constraints as possible. In principle, this "best" partial solution could be translated to annotations, but it would be deeply tied to a particular constraint generation scheme. Our definition of best annotations (section 4.2) is independent of any particular static checking technique, so it is a better basis for comparing tools.

## 7 Conclusion

Type reconstruction experiments systematically overestimate the effectiveness of nullability annotation inference tools, because developers make semantic changes that make both checking and inference simpler when annotating their code. We therefore conducted experiments using a new methodology, based on a novel theoretical definition of which nullability annotations are "best," to fairly compare three extant nullability annotation inference tools. Our results suggest that while Nullaway Annotator is the best tool overall, all three have distinct strengths and weaknesses, and all three can improve a lot. Our work points towards how to improve these tools along several dimensions: make inference tools suggest code changes, not just annotations; define a fitness function that captures developer intent better than "minimize checker errors"; and more.

*Data availability.* Our scripts, data, and records of human judgments are available online [Karimipour et al. 2024].

## Acknowledgments

This research was supported in part by the National Science Foundation under grants CCF-2223826, CCF-2312262, and CCF-2312263, a gift from Oracle Labs, and a Google Research Award.

## References

Subarno Banerjee, Lazaro Clapp, and Manu Sridharan. 2019. NullAway: Practical type-based null safety for Java. In *ESEC/FSE 2019: The ACM 27th joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. Tallinn, Estonia, 740–750.

CF Developers. 2024. Nullness Checker. https://checkerframework.org/manual/#nullness-checker. Accessed 2024-09-11.

Jacob Cohen. 1960. Cohen's kappa. https://en.wikipedia.org/wiki/Cohen%27s_kappa Accessed: 2024-12-03.

Werner Dietl, Stephanie Dietzel, Michael D. Ernst, Kıvanç Muşlu, and Todd Schiller. 2011. Building and using pluggable type-checkers. In *ICSE 2011, Proceedings of the 33rd International Conference on Software Engineering*. Waikiki, Hawaii, USA, 681–690.

Jens Dietrich, David J. Pearce, and Mahin Chandramohan. 2023. On Leveraging Tests to Infer Nullable Annotations. In *37th European Conference on Object-Oriented Programming (ECOOP 2023) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 263)*, Karim Ali and Guido Salvaneschi (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 10:1–10:25. https://doi.org/10.4230/LIPIcs.ECOOP.2023.10

Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. 2007. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming* 69, 1–3 (Dec. 2007), 35–45.

Google. 2024a. Android Developers: Activity documentation. https://developer.android.com/reference/android/app/Activity. Accessed: 2024-09-10.

Google. 2024b. Android Developers: Activity lifecycle. https://developer.android.com/reference/android/app/Activity#activity-lifecycle. Accessed: 2024-09-10.

James Gosling, Bill Joy, Guy Steele, Gilad Bracha, Alex Buckley, Daniel Smith, and Gavin Bierman. 2021. The Java Language Specification, Java SE 17 Edition, Section 12.5: Creation of New Class Instances. https://docs.oracle.com/javase/specs/jls/se17/html/jls-12.html#jls-12.5.

David Hovemeyer, Jaime Spacco, and William Pugh. 2005. Evaluating and tuning a static analysis to find null pointer bugs. In *PASTE 2005: ACM SIGPLAN/SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE 2005)*. Lisbon, Portugal, 13–19.

JetBrains. 2006. Detecting probable NPE's. https://blog.jetbrains.com/idea/2006/03/detecting-probably-npes/. Accessed 9 September 2024.

Christian Gram Kalhauge and Jens Palsberg. 2019. Binary reduction of dependency graphs. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 556–566.

Nima Karimipour, Erfan Arvan, Martin Kellogg, and Manu Sridharan. 2024. Nullability Inference Tools Evaluation Experiments. https://github.com/erfan-arvan/nullability-inference-comparison-tools-scripts.

Nima Karimipour, Justin Pham, Lazaro Clapp, and Manu Sridharan. 2023. Practical Inference of Nullability Types. In *31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2023)*. https://doi.org/10.1145/3611643.3616326

Martin Kellogg, Daniel Daskiewicz, Loi Ngo Duc Nguyen, Muyeed Ahmed, and Michael D. Ernst. 2023. Pluggable type inference for free. In *ASE 2023: Proceedings of the 38th Annual International Conference on Automated Software Engineering*. Luxembourg, 1542–1554.

Thanh Le-Cong, Hong Jin Kang, Truong Giang Nguyen, Stefanus Agus Haryono, David Lo, Xuan-Bach D Le, and Quyet Thang Huynh. 2022. Autopruner: transformer-based call graph pruning. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 520–532.

Yi Li, Aashish Yadavally, Jiaxing Zhang, Shaohua Wang, and Tien N Nguyen. 2023. DeMinify: Neural Variable Name Recovery and Type Inference. In *Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 758–770.

Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondřej Lhoták, J. Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z. Guyer, Uday P. Khedker, Anders Møller, and Dimitrios Vardoulakis. 2015. In defense of soundiness: a manifesto. *Commun. ACM* 58, 2 (Jan. 2015), 44–46. https://doi.org/10.1145/2644805

Alexey Loginov, Eran Yahav, Satish Chandra, Stephen Fink, Noam Rinetzky, and Mangala Nanda. 2008. Verifying dereference safety via expanding-scope analysis. In *International Symposium on Software Testing and Analysis (ISSTA)*. Association for Computing Machinery (ACM), 213–224.

Calvin Loncaric, Satish Chandra, Cole Schlesinger, and Manu Sridharan. 2016. A Practical Framework for Type Inference Error Explanation. *SIGPLAN Not.* 51, 10 (oct 2016), 781–799. https://doi.org/10.1145/3022671.2983994

Ravichandhran Madhavan and Raghavan Komondoor. 2011. Null dereference verification via over-approximated weakest pre-conditions analysis. *ACM Sigplan Notices* 46, 10 (2011), 1033–1052.

Amir M Mir, Mehdi Keshani, and Sebastian Proksch. 2024. On the Effectiveness of Machine Learning-based Call Graph Pruning: An Empirical Study. In *2024 IEEE/ACM 21st International Conference on Mining Software Repositories (MSR)*. IEEE, 457–468.

Amir M Mir, Evaldas Latoškinas, and Georgios Gousios. 2021. ManyTypes4Py: A benchmark python dataset for machine learning-based type inference. In *International Conference on Mining Software Repositories (MSR)*. IEEE, 585–589.

Amir M Mir, Evaldas Latoškinas, Sebastian Proksch, and Georgios Gousios. 2022. Type4Py: Practical deep similarity learning-based type inference for Python. In *International Conference on Software Engineering (ICSE)*. 2241–2252.

Oracle. 2015. The Java Language Specification, Java SE 8 Edition, Section 17.4: Memory Model. https://docs.oracle.com/javase/specs/jls/se8/html/jls-17.html#jls-17.4 Accessed: 2024-12-03.

Matthew M. Papi, Mahmood Ali, Telmo Luis Correa Jr., Jeff H. Perkins, and Michael D. Ernst. 2008. Practical pluggable types for Java. In *ISSTA 2008, Proceedings of the 2008 International Symposium on Software Testing and Analysis*. Seattle, WA, USA, 201–212. https://doi.org/10.1145/1390630.1390656

Zvonimir Pavlinovic, Tim King, and Thomas Wies. 2014. Finding Minimum Type Error Sources. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)* (Portland, Oregon, USA) *(OOPSLA '14)*. Association for Computing Machinery, New York, NY, USA, 525–542. https://doi.org/10.1145/2660193.2660230

Yun Peng, Cuiyun Gao, Zongjie Li, Bowei Gao, David Lo, Qirun Zhang, and Michael Lyu. 2022. Static inference meets deep learning: a hybrid type inference approach for Python. In *ICSE 2022, Proceedings of the 43rd International Conference on Software Engineering*. Pittsburgh, PA, USA, 2019–2030. https://doi.org/10.1145/3510003.3510038

Yun Peng, Chaozheng Wang, Wenxuan Wang, Cuiyun Gao, and Michael R Lyu. 2023. Generative type inference for Python. In *International Conference on Automated Software Engineering (ASE)*. IEEE, 988–999.

Artem Pianykh, Ilya Zorin, and Dmitry Lyubarskiy. 2022. Retrofitting null-safety onto Java at Meta. https://engineering.fb.com/2022/11/22/developer-tools/meta-java-nullsafe/.

Benjamin C. Pierce. 2002. *Types and Programming Languages* (1st ed.). The MIT Press.

William Pugh et al. 2008. FindBugs: A Program to find Bugs in Java Programs. http://findbugs.sourceforge.net/.

Henry Gordon Rice. 1953. Classes of recursively enumerable sets and their decision problems. *Trans. Amer. Math. Soc.* 74, 2 (1953), 358–366.

Kazi Siddiqui and Martin Kellogg. 2024. Inferring Pluggable Types with Machine Learning. https://arxiv.org/abs/2406.15676.

Alexander J. Summers and Peter Müller. 2011. Freedom before commitment: A lightweight type system for object initialisation. In *OOPSLA 2011, Object-Oriented Programming Systems, Languages, and Applications*. Portland, OR, USA, 1013–1032. https://doi.acm.org/10.1145/2048066.2048142

The Daikon developers. 2024. Daikon Invariant Detector User Manual version 5.8.2, section 8.1.5: AnnotateNullable. http://plse.cs.washington.edu/daikon/download/doc/daikon.html#AnnotateNullable. Accessed 9 September 2024.

Akshay Utture, Christian Gram Kalhauge, Shuyang Liu, and Jens Palsberg. 2020. NJR-1 dataset. https://zenodo.org/records/8015477.

Akshay Utture, Shuyang Liu, Christian Gram Kalhauge, and Jens Palsberg. 2022. Striking a balance: pruning false-positives from static call graphs. In *Proceedings of the 44th International Conference on Software Engineering*. 2043–2055.

Akshay Utture and Jens Palsberg. 2023. From Leaks to Fixes: Automated Repairs for Resource Leak Warnings. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 159–171.

Rijnard van Tonder and Claire Le Goues. 2020. Tailoring programs for static analysis via program transformation. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 824–834.

Danfeng Zhang, Andrew C. Myers, Dimitrios Vytiniotis, and Simon Peyton-Jones. 2017. SHErrLoc: A Static Holistic Error Locator. *ACM Trans. Program. Lang. Syst.* 39, 4, Article 18 (aug 2017), 47 pages. https://doi.org/10.1145/3121137

Received 2024-09-13; accepted 2025-01-14