

Gradual Program Analysis for Null Pointers

Sam Estep

Carnegie Mellon University, Pittsburgh, PA, USA

Jenna Wise

Carnegie Mellon University, Pittsburgh, PA, USA

Jonathan Aldrich

Carnegie Mellon University, Pittsburgh, PA, USA

Éric Tanter

Computer Science Department (DCC), University of Chile, Santiago, Chile

Johannes Bader

Jane Street, New York, NY, USA

Joshua Sunshine

Carnegie Mellon University, Pittsburgh, PA, USA

Abstract

Static analysis tools typically address the problem of excessive false positives by requiring programmers to explicitly annotate their code. However, when faced with incomplete annotations, many analysis tools are either too conservative, yielding false positives, or too optimistic, resulting in unsound analysis results. In order to flexibly and soundly deal with partially-annotated programs, we propose to build upon and adapt the gradual typing approach to abstract-interpretation-based program analyses. Specifically, we focus on null-pointer analysis and demonstrate that a gradual null-pointer analysis hits a sweet spot, by gracefully applying static analysis where possible and relying on dynamic checks where necessary for soundness. In addition to formalizing a gradual null-pointer analysis for a core imperative language, we build a prototype using the Infer static analysis framework, and present preliminary evidence that the gradual null-pointer analysis reduces false positives compared to two existing null-pointer checkers for Infer. Further, we discuss ways in which the gradualization approach used to derive the gradual analysis from its static counterpart can be extended to support more domains. This work thus provides a basis for future analysis tools that can smoothly navigate the tradeoff between human effort and run-time overhead to reduce the number of reported false positives.

2012 ACM Subject Classification Software and its engineering → General programming languages; Software and its engineering → Software verification

Keywords and phrases gradual typing, gradual verification, dataflow analysis

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2021.3

Related Version *Full Version*: <https://arxiv.org/abs/2105.06081> [10]

Supplementary Material *Software*: <https://github.com/orgs/gradual-verification/packages/container/package/ecoop21>

Funding National Science Foundation under Grant No. CCF-1901033 and Grant No. CCF-1852260
Éric Tanter: FONDECYT Regular project 1190058

1 Introduction

Static analysis is useful [1], but underused in practice because of false positives [15]. A commonly-used way to reduce false positives is through programmer-provided annotations [4] that make programmers intent manifest. For example, Facebook’s Infer Eradicate [11], Uber’s NULLAWAY [3], and the Java Nullness Checker from the Checker Framework [21] all rely on `@NonNull` and `@Nullable` annotations to statically find and report potential null-pointer



© Sam Estep, Jenna Wise, Jonathan Aldrich, Éric Tanter, Johannes Bader, and Joshua Sunshine; licensed under Creative Commons License CC-BY 4.0

35th European Conference on Object-Oriented Programming (ECOOP 2021).

Editors: Manu Sridharan and Anders Møller; Article No. 3; pp. 3:1–3:25

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

exceptions in Java code. However, in practice, annotating code completely can be very costly [6] – or even impossible, for instance, when relying on third-party libraries and APIs. As a result, since non-null reference variables are used extensively in software [6], many tools assume missing annotations are `@NonNull`. But, the huge number of false positives produced by such an approach in practice is a serious burden. To address this pitfall, `NULLAWAY` assumes that sinks (i.e. targets of assignments and bindings) are `@Nullable` and sources are `@NonNull`. Unfortunately, both strategies are unsound, and therefore programs deemed valid may still raise null pointer exceptions at run time.

This paper explores a novel approach to these issues by drawing on research in gradual typing [22, 23, 14] and its recent adaptation to gradual verification [2, 24]. We propose gradual program analysis as a principled, sound, and practical way to handle missing annotations. As a first step in the research agenda of gradual program analysis, this article studies the case of a simple null-pointer analysis. We present a general formal framework to derive gradual program analyses by transforming static analyses based on abstract interpretation [8]. Specifically, we study analyses that operate over first-order procedural imperative languages and support user-provided annotations. This setting matches the core language used by many tools, such as Infer. In essence, a *gradual analysis* treats missing annotations optimistically, but injects run-time checks to preserve soundness. Crucially, the static portion of a gradual analysis uses the same algorithmic architecture as the underlying static analysis.¹

Additionally, we ensure that any gradual analysis produced from our framework satisfies the *gradual guarantees*, adapted from Siek *et al.* [23] formulation for gradual typing. Any gradual analysis is also a *conservative extension* of the base static analysis: when all annotations are provided, the gradual analysis is equivalent to the base static analysis, and no run-time checks are inserted. Therefore, the gradual analysis smoothly trades off between static and dynamic checking, driven by the annotation effort developers are willing to invest.

To provide initial evidence of the applicability of gradual null-pointer analysis, we implement a gradual null-pointer analysis (GNPA) using Facebook’s Infer analysis framework and report on preliminary experiments using the prototype.² The experiments show that a gradual null-pointer analysis can be effectively implemented, and used at scale to produce a small number of false positives in practice – fewer than Infer `ERADICATE` as well as a more recent Infer checker, `NULLSAFE`. They also show that GNPA eliminates on average more than half of the null-pointer checks Java automatically inserts at run time. As a result, unlike other null-pointer analyses, GNPA can both prove the redundancy of run-time checks and reduce reported false positives.

The rest of the paper is organized as follows. In Section 2, we motivate gradual program analysis in the setting of null pointers by looking at how `ERADICATE`, `NULLSAFE`, `NULLAWAY`, and the Java Nullness Checker operate on example code with missing annotations, showcasing the concrete advantages of GNPA. Section 3 formalizes `PICL`, a core imperative language similar to that of Infer. Section 4 then presents the static null-pointer analysis (NPA) for `PICL`, which is then used as the starting point for the derivation of the gradual analysis. We describe our approach to gradualizing a static program analysis in Section 5, using GNPA as the running case study. Additionally, Section 5 includes a discussion of important gradual properties our analysis adheres to: *soundness*, *conservative extension*, and the *gradual guarantee*. All proofs can be found in the appendix of the full version of this paper [10]. We

¹ Note that an alternative is phrasing nullness as a type system, which can also be gradualized [5, 19]. We focus on approaches based on static analysis, which have very different technical foundations and user experience. We compare to type-based approaches in Section 7.

² <https://github.com/orgs/gradual-verification/packages/container/package/ecoop21>

report on the preliminary empirical evaluation of an Infer GNPA checker called *Graduator* in Section 6. Section 7 discusses related work and Section 8 concludes. In the conclusion, we sketch ways in which the approach presented here could be applied to other analysis domains, highlight open venues for future work in the area of gradual program analysis.

2 Gradual Null-Pointer Analysis in Action

This section informally introduces gradual null-pointer analysis and its potential compared to existing approaches through a simple example. We first briefly recall the basics of null-pointer analyses, and then discuss how current tools deal with missing annotations in problematic ways.

2.1 Null-Pointer Analysis in a Nutshell

With programming languages that allow any type to be inhabited by a null value, programmers end up facing runtime errors (or worse if the language is unsafe) related to dereferencing null pointers. A null-pointer analysis is a static analysis that detects *potential* null pointer dereferences and reports them as warnings, so that programmers can understand where explicit nullness checks should be added in order to avoid runtime errors. Examples of null-pointer analyses are Infer Eradicate [12] and the Java Null Checker [21]. Typically, a null-pointer analysis allows programmers to add annotations in the code to denote which variables (as well as fields, return values, etc.) are, or should be, non-null—e.g. `@NonNull`—and which are potentially null—e.g. `@Nullable`. A simple flow analysis is then able to detect and report conflicts, such as when a nullable variable is assigned to a non-null variable.

While a static null pointer analysis brings guarantees of robustness to a codebase, its adoption is not necessarily seamless. If a static analysis aims to be sound, it must not suffer from false negatives, i.e. miss any actual null pointer dereference that can happen at runtime. While desirable, this means the analysis necessarily has to be conservative and therefore reports false positives – locations that are thought to potentially trigger null pointer dereferences, but actually do not.

This standard static analysis conundrum is exacerbated when considering programs where not all variables are annotated. Of course, in practice, a codebase is rarely fully annotated. Existing null-pointer analyses assign missing annotations a concrete annotation, such as `Nullable` or `NonNull`. In doing so, they either report additional false positives, suffer from false negatives (and hence are unsound), or both. The rest of this section illustrates these issues with a simple example, and discusses how a gradual null-pointer analysis (GNPA) alleviates them. GNPA treats missing annotations in a special manner, following the gradual typing motto of being optimistic statically and relying on runtime checks for soundness [22]. Doing so allows the analysis to leverage both static and dynamic information to reduce false positives while maintaining soundness.

2.2 Avoiding False Positives

GNPA can reduce the number of false positives reported by static tools by leveraging provided annotations and run-time checks. We demonstrate this with the unannotated program in Figure 1. The program appends the reverse of a non-null string to the reverse of a null string and prints the result. The `reverse` method (lines 3–8) returns the reverse of an input string when it is non-null and an empty string when the input is `null`. Additionally, `reverse` is unannotated, as highlighted for reference.

3:4 Gradual Program Analysis for Null Pointers

```
1 class Main {
2
3     static String reverse(String str) {
4         if (str == null) return new String();
5         StringBuilder builder = new StringBuilder(str);
6         builder.reverse();
7         return builder.toString();
8     }
9
10    public static void main(String[] args) {
11        String reversed = reverse(null);
12        String frown = reverse(":)");
13        String both = reversed.concat(frown);
14        System.out.println(both);
15    }
16 }
```

■ **Figure 1** Unannotated Java code safely reversing nullable strings.

The most straightforward approach to handling the missing annotations is to replace them with a fixed annotation. Infer Eradicate and the Java Nullness Checker both choose `@NonNull` as the default, since that is the most frequent annotation used in practice [6]. Thus, in this example, they would treat `reverse`'s argument and return value as annotated with `@NonNull`. This correctly assigns `reversed` and `frown` as non-null on lines 11 and 12; and consequently, no false positive is reported when `reversed` is dereferenced on line 13. However, both tools will report a false positive each time `reverse` is called with `null`, as in line 11.

Other uniform defaults are possible, but likewise lead to false positive warnings. For example, choosing `@Nullable` by default would result in a false positive when `reversed` is dereferenced. A more sophisticated choice would be the Java Nullness Checker's `@PolyNull` annotation, which supports type qualifier polymorphism for methods annotated with `@PolyNull`. If `reverse`'s method signature is annotated with `@PolyNull`, then `reverse` would have two conceptual versions:

```
static @Nullable String reverse(@Nullable String str)
static @NonNull String reverse(@NonNull String str)
```

At a call site, the most precise applicable signature would be chosen; so, calling `reverse` with `null` (line 11) would result in the `@Nullable` signature, and calling `reverse` with `":)"` (line 12) would result in the `@NonNull` signature. Unfortunately, this strategy marks `reversed` on line 11 as `@Nullable` even though it is `@NonNull`, and a false positive is reported when `reversed` is dereferenced on line 13. So while `@PolyNull` increases the expressiveness of the annotation system, it does not solve the problem of avoiding false positives from uniform annotation defaults.

In contrast, GNPA optimistically assumes both calls to `reverse` in `main` (lines 11–12) are valid without assigning fixed annotations to `reverse`'s argument or return value. Then, the analysis can continue relying on *contextual optimism* when reasoning about the rest of `main`: `reversed` is assumed `@NonNull` to satisfy its dereference on line 13. Of course this is generally an unsound assumption, so a run-time check is inserted to ascertain the

non-nullness of `reversed` and preserve soundness. Alternatively, a developer could annotate the return value of `reverse` with `@NonNull`. GNPA will operate as before except it will leverage this new information during static reasoning. Therefore, `reversed` will be marked `@NonNull` on line 11 and the dereference of `reversed` on line 13 will be statically proven safe without any run-time check.

It turns out that a non-uniform choice of defaults can be optimistic in the same sense as GNPA. For example, `NULLAWAY` assumes sinks are `@Nullable` and sources are `@NonNull` when annotations are missing. In fact, this strategy correctly annotates `reverse`, and so no false positives are reported by the tool for the program in Figure 1. However, in contrast to the gradual approach, the `NULLAWAY` approach is in fact unsound, as illustrated next.

2.3 Avoiding False Negatives

When Eradicate, `NULLAWAY`, and the Java Nullness Checker handle missing annotations, they all give up soundness in an attempt to limit the number of false positives produced.

To illustrate, consider the same program from Figure 1, with one single change: the `reverse` method now returns `null` instead of an empty string (line 4).

```
if (str == null) return null;
```

All of the tools mentioned earlier, including `NULLAWAY`, erroneously assume that the return value of `reverse` is `@NonNull`. On line 11, `reversed` is assigned `reverse(null)`'s return value of `null`; so, it is an error to dereference `reversed` on line 13. Unfortunately, all of the tools assume `reversed` is assigned a non-null value and do not report an error on line 13. This is a *false negative*, which means that at runtime the program will fail with a null-pointer exception.

GNPA is similarly optimistic about `reversed` being non-null on line 13. However, GNPA safeguards its optimistic static assumptions with run-time checks. Therefore, the analysis will correctly report an error on line 13. Alternatively, a developer could annotate the return value of `reverse` with `@Nullable`. By doing so, the gradual analysis will be able to exploit this information statically to report a static error, instead of a dynamic error.

To sum up, a gradual null-pointer analysis can reduce false positives by optimistically treating missing annotations, and preserve soundness by detecting errors at runtime. Of course, one may wonder why it is better to fail at runtime when passing a null value as a non-null annotated argument, instead of just relying on the upcoming null-pointer exception. There are two answers to this question. First, in unsafe languages like C, a null-pointer dereference results in a crash. Second, in a safe language like Java where a null-pointer dereference is anyway detected and reported, it can be preferable to fail as soon as possible, in order to avoid performing computation (and side effects) under an incorrect assumption. This is similar to how the eager reporting of gradual typing can be seen as an improvement over simply relying on the underlying safety checks of a dynamically-typed language.

Next, we formally develop GNPA, prove that it is sound, and prove that it smoothly trades-off between static and dynamic checking following the gradual guarantee criterion from gradual typing [23]. We finally report on an actual implementation of GNPA and compare its effectiveness with existing tools.

$x, y \in \text{VAR}$ $e \in \text{EXPR}$ $a \in \text{ANN} = \{\text{Nullable}, \text{NonNull}, ?\}$ $P ::= \overline{\text{procedure field } s}$ $\text{field} ::= T f;$ $\text{procedure} ::= T@a m (\overline{T@a x}) \{ s \}$ $T ::= \text{ref}$ $\oplus ::= \wedge \mid \vee$	$m \in \text{PROC}$ $f \in \text{FIELD}$ $s \in \text{STMT}$ $e ::= \text{null} \mid x \mid e \oplus e \mid e.f \mid \text{new}(\overline{f})$ $\quad \mid m(x)$ $c ::= e = \text{null} \mid e \neq \text{null}$ $s ::= \text{skip} \mid s ; s \mid T x \mid x := e$ $\quad \mid x.f := y \mid \text{if } (c) \{ s \} \text{ else } \{ s \}$ $\quad \mid \text{while } (c) \{ s \} \mid \text{return } y$
--	---

■ **Figure 2** Abstract syntax of PICL.

3 PICL: A Procedural Imperative Core Language

Following the Abstract Gradual Typing methodology introduced by Garcia *et al.* [14], we build GNPA on top of a static null-pointer analysis, NPA. Thus, we first formally present a procedural imperative core language (PICL), used for both analyses to operate on; we present NPA in Section 4, and GNPA in Section 5. PICL is akin to the intermediate language of the Infer framework, and therefore the formal development around PICL drove the implementation of the Infer GNPA checker we evaluate in Section 6.

3.1 Syntax & Static Semantics

The syntax of PICL can be found in Figure 2. Programs consist of procedures³, fields, and statements. Statements include the empty statement, sequences, variable declarations, variable and field assignments, conditionals, while loops, and returns. Expressions consist of null literals, variables, comparisons, conjunctions, disjunctions, field accesses, object allocations, and procedure calls. Finally, procedures may have `Nullable` or `NonNull` annotations on their arguments and return values. Missing annotations are represented by `?`.

As the focus of this work is not on typing, we only consider well-formed and well-typed programs, which is standard and not formalized here. In particular, variables are declared and initialized before use, and field and procedure names are unique.

3.2 Control Flow Graph Representation

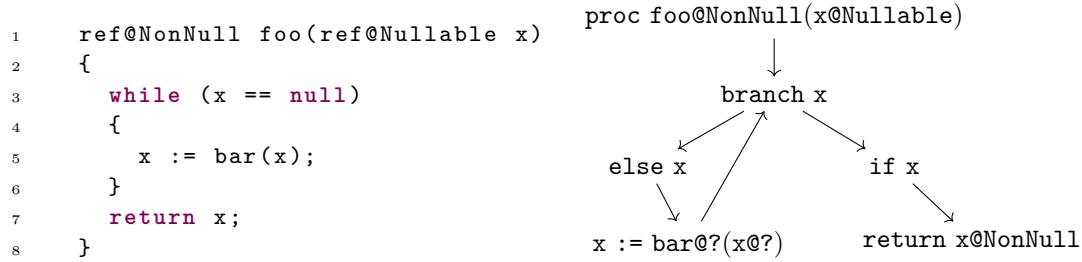
Well-formed programs written in the abstract syntax given in Fig. 2 are translated into *control flow graphs* – one graph for each procedure body and one for the main s . A finite control flow graph (CFG) for program p has vertices VERT_p and edges $\text{EDGE}_p \subseteq \text{VERT}_p \times \text{VERT}_p$. For $v_1, v_2 \in \text{VERT}_p$, we write $v_1 \xrightarrow{p} v_2$ to denote $(v_1, v_2) \in \text{EDGE}_p$. Each vertex holds a single instruction, which we can access using the function $\text{INST}_p : \text{VERT}_p \rightarrow \text{INST}$. We write $[\iota]_v$ to denote a vertex $v \in \text{VERT}_p$ such that $\text{INST}_p(v) = \iota$, or just $[\iota]$ (omitting the v) when the vertex itself is not important. By construction, these translated CFGs satisfy certain well-formedness properties, listed in the appendix of the full version of this paper [10].

The set of possible instructions is defined in Figure 3. In general, the CFG instructions are atomic variants of program statements designed to simplify the analysis presentations. Figure 4 gives the CFG of a simple procedure `foo`, which calls `bar` repeatedly until `x` becomes non-null

³ Procedures accept only one parameter to simplify later formalisms.

$$\begin{array}{ll}
x, y, z \in \text{VAR} & m \in \text{PROC} \\
a, b \in \text{ANN} = \{\text{Nullable}, \text{NonNull}, ?\} & f \in \text{FIELD} \\
I ::= x := y \mid x := \text{null} \mid x := m@a(y@b) \mid x := \text{new}(\overline{f}) \mid x := y \wedge z \mid x := y \vee z \\
\quad \mid x := y.f \mid x.f := y \mid \text{branch } x \mid \text{if } x \mid \text{else } x \mid \text{return } y@a \mid \text{main} \\
\quad \mid \text{proc } m@a(y@b)
\end{array}$$

■ **Figure 3** Abstract syntax of a CFG instruction.



■ **Figure 4** Example CFG.

and then returns x . The CFG starts with foo 's entry node $\text{proc } \text{foo@NonNull}(x@Nullable)$ (similarly, main is always the entry node of the main program's CFG). Then, the while loop on lines 3–6 results in the $\text{branch } x$ sub-graph, which leads to $\text{if } x$ when x is non-null and $\text{else } x$ when x is null. The call to bar follows from $\text{else } x$ and loops back to $\text{branch } x$ as expected. Finally, $\text{return } x@NonNull$ follows from $\text{if } x$ ending the CFG. Precise semantics for instructions is given in Section 3.3.

3.3 Dynamic Semantics

We define the set of possible object locations as the set of natural numbers and 0, $\text{VAL} = \mathbb{N} \cup \{0\}$. The null pointer is location 0.

Now, a program state $(\text{STATE}_p \subseteq \text{STACK}_p \times \text{MEM}_p)$ consists of a stack and a heap. A heap $\mu \in \text{MEM}_p = (\text{VAL} \setminus \{0\}) \rightarrow (\text{FIELD} \rightarrow \text{VAL})$ maps object locations and field names to program values – other (possibly null) pointers. A stack is made of stack frames each containing a local variable environment and CFG node:

$$\begin{array}{l}
S \in \text{STACK}_p ::= E \cdot S \mid \text{nil} \quad \text{where} \quad E \in \text{FRAME}_p = \text{ENV} \times \text{VERT}_p \\
\quad \text{and} \quad \text{ENV} = \text{VAR} \rightarrow \text{VAL}.
\end{array}$$

Further, we restrict the set of states $\xi = \langle \langle \rho_1, v_1 \rangle \cdot \langle \rho_2, v_2 \rangle \cdots \langle \rho_n, v_n \rangle \cdot \text{nil} \parallel \mu \rangle \in \text{STATE}_p$ to include only those satisfying the following conditions:

1. *Bottom stack frame is in main:* Let $\text{DESCEND} : \text{VERT}_p \rightarrow \mathcal{P}^+(\text{VERT}_p)$ give the descendants of each node in the control flow graph. Then $v_i \in \text{DESCEND}(v_0)$ if and only if $i = n$.
2. *Every variable defaults to null (except on main and proc nodes):* If $\text{INST}_p(v_i) \neq \text{main}$ and $\text{INST}_p(v_i) \neq \text{proc } m@a(y@b)$ then ρ_i is a total function.
3. *Follow the “true” branch when non-null:* If $\text{INST}_p(v_i) = \text{if } y$ then $\rho_i(y) \neq 0$.
4. *Follow the “false” branch when null:* If $\text{INST}_p(v_i) = \text{else } y$ then $\rho_i(y) = 0$.

$$\begin{aligned}
 & \langle \langle \rho, [x := y]_u \rangle \cdot S \parallel \mu \rangle \longrightarrow_p \langle \langle \rho[x \mapsto \rho(y)], v \rangle \cdot S \parallel \mu \rangle \\
 & \langle \langle \rho, [\text{branch } y]_u \rangle \cdot S \parallel \mu \rangle \longrightarrow_p \langle \langle \rho, [\text{BRANCH}(\rho(y), y)]_v \rangle \cdot S \parallel \mu \rangle \\
 & \langle \langle \rho, [\text{if } y]_u \rangle \cdot S \parallel \mu \rangle \longrightarrow_p \langle \langle \rho, v \rangle \cdot S \parallel \mu \rangle \\
 & \langle \langle \rho, [\text{else } y]_u \rangle \cdot S \parallel \mu \rangle \longrightarrow_p \langle \langle \rho, v \rangle \cdot S \parallel \mu \rangle \\
 & \langle \langle \rho, [x := m@a(y@b)]_u \rangle \cdot S \parallel \mu \rangle \longrightarrow_p \langle \langle \emptyset, [\text{proc } m@a(y'@b)] \rangle \cdot \langle \rho, u \rangle \cdot S \parallel \mu \rangle \\
 & \langle \langle \rho_1, [\text{proc } m@a(y@b)]_u \rangle \cdot \langle \rho_2, [x := m@a(y'@b)]_w \rangle \cdot S \parallel \mu \rangle \longrightarrow_p \langle \langle \rho_0[y \mapsto \rho_2(y')], v \rangle \cdot \langle \rho_2, w \rangle \cdot S \parallel \mu \rangle \\
 & \langle \langle \rho_1, [\text{return } y@a] \rangle \cdot \langle \rho_2, [x := m@a(y'@b)]_u \rangle \cdot S \parallel \mu \rangle \longrightarrow_p \langle \langle \rho_2[x \mapsto \rho_1(y)], v \rangle \cdot S \parallel \mu \rangle \dagger \\
 & \langle \langle \rho, [x := \text{null}]_u \rangle \cdot S \parallel \mu \rangle \longrightarrow_p \langle \langle \rho[x \mapsto 0], v \rangle \cdot S \parallel \mu \rangle \\
 & \langle \langle \rho, [x := \text{new}(\bar{f})]_u \rangle \cdot S \parallel \mu \rangle \longrightarrow_p \langle \langle \rho[x \mapsto \text{NEW}(\mu)], v \rangle \cdot S \parallel \mu[\text{NEW}(\mu) \mapsto \overline{[f_i \mapsto \text{null}]}] \rangle \\
 & \langle \langle \rho, [x := y \wedge z]_u \rangle \cdot S \parallel \mu \rangle \longrightarrow_p \langle \langle \rho[x \mapsto \text{AND}(\rho(y), \rho(z))], v \rangle \cdot S \parallel \mu \rangle \\
 & \langle \langle \rho, [x := y \vee z]_u \rangle \cdot S \parallel \mu \rangle \longrightarrow_p \langle \langle \rho[x \mapsto \text{OR}(\rho(y), \rho(z))], v \rangle \cdot S \parallel \mu \rangle \\
 & \langle \langle \rho, [x := y.f]_u \rangle \cdot S \parallel \mu \rangle \longrightarrow_p \langle \langle \rho[x \mapsto \mu(\rho(y))(f)], v \rangle \cdot S \parallel \mu \rangle \\
 & \langle \langle \rho, [x.f := y]_u \rangle \cdot S \parallel \mu \rangle \longrightarrow_p \langle \langle \rho, v \rangle \cdot S \parallel \mu[\rho(x) \mapsto [f \mapsto \rho(y)]] \rangle \\
 & \langle \langle \rho, [\text{main}]_u \rangle \cdot S \parallel \mu \rangle \longrightarrow_p \langle \langle \rho_0, v \rangle \cdot S \parallel \mu \rangle
 \end{aligned}$$

■ **Figure 5** Small-step semantics rules that hold when $u \xrightarrow{p} v$. † This particular rule only applies if either $a = ?$ or $\rho_1(y) \in \text{CONC}(a)$ (see Section 4).

5. *Every frame except the top is a procedure call:* If $v_i \in \text{DESCEND}(\text{proc } m@a(y@b))$ then $\text{INST}_p(v_{i+1}) = x := m@a(y'@b)$, and either $b = ?$ or $\rho_{i+1}(y') \in \text{CONC}(b)$ (see section 4).

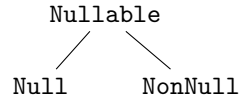
Now, the small-step semantics of PICL is given in Figure 5, where $\rho_0 = \{x \mapsto 0 : x \in \text{VAR}\}$. The rules rely on the following helper functions:

$$\begin{array}{ll}
 \text{NEW} : \text{MEM}_p \rightarrow \text{VAL} \setminus \{0\} & \text{NEW}(\mu) = 1 + \max(\{0\} \cup \text{dom}(\mu)) \\
 \text{BRANCH} : \text{VAL} \times \text{VAR} \rightarrow \text{INST} & \text{BRANCH}(n, x) = \text{if } x \text{ if } n > 0; \text{ else } x \text{ otherwise} \\
 \text{AND} : \text{VAL} \times \text{VAL} \rightarrow \text{VAL} & \text{AND}(n_1, n_2) = n_2 \text{ if } n_1 > 0; n_1 \text{ otherwise} \\
 \text{OR} : \text{VAL} \times \text{VAL} \rightarrow \text{VAL} & \text{OR}(n_1, n_2) = n_1 \text{ if } n_1 > 0; n_2 \text{ otherwise}
 \end{array}$$

Notably, `branch` y steps to the `if` y node when y is non-null and `else` y when y is null. Additionally, if a procedure call’s argument disagrees with its parameter annotation, then it will get stuck (rule 5 for states); otherwise, the call statement will safely step to the procedure’s body. In contrast, the semantics will get stuck if a return value does not agree with the procedure’s return annotation.

4 A Static Null-Pointer Analysis for PICL

In this section, we formalize a static null-pointer analysis, called NPA, for PICL on which we will build GNPA. Here, we will only consider completely annotated programs, $\text{ANN} = \{\text{Nullable}, \text{NonNull}\}$. Therefore, we use a “prime” symbol for sets like $\text{INST}' \subseteq \text{INST}$ to indicate that this is not the whole story. We present NPA’s semilattice of abstract values, flow function, fixpoint algorithm, and how the analysis uses the results from the fixpoint algorithm to report warnings to the user.



■ **Figure 6** The ABST semilattice.

4.1 Semilattice of Abstract Values

The set of abstract values $\text{ABST} = \{\text{Nullable}, \text{Null}, \text{NonNull}\}$ make up the finite semilattice defined in Figure 6. The partial order $\sqsubseteq \subseteq \text{ABST} \times \text{ABST}$ given is

$$\text{Null} \sqsubseteq \text{Nullable} \quad \text{NonNull} \sqsubseteq \text{Nullable} \quad \forall. l \in \text{ABST} . l \sqsubseteq l.$$

The join function $\sqcup : \text{ABST} \times \text{ABST} \rightarrow \text{ABST}$ induced by the partial order is:

$$\begin{aligned} \text{Null} \sqcup \text{NonNull} &= \text{Nullable} & \forall. l \in \text{ABST} . l \sqcup \text{Nullable} &= \text{Nullable} \\ \forall. l \in \text{ABST} . l \sqcup l &= l \end{aligned}$$

Clearly, **Nullable** is the top element \top . Next, we relate this semilattice to **VAL** via a concretization function $\text{CONC} : \text{ABST} \rightarrow \mathcal{P}^+(\text{VAL})$:

$$\text{CONC}(\text{Nullable}) = \text{VAL}, \quad \text{CONC}(\text{Null}) = \{0\}, \quad \text{CONC}(\text{NonNull}) = \text{VAL} \setminus \{0\},$$

which satisfies the property $\forall. l_1, l_2 \in \text{ABST} . l_1 \sqsubseteq l_2 \iff \text{CONC}(l_1) \subseteq \text{CONC}(l_2)$.

4.2 Flow Function

Similar to how we use **ENV** to represent mappings from variables to concrete values, we will use $\sigma \in \text{MAP} = \text{VAR} \rightarrow \text{ABST}$ to represent mappings from variables to abstract values – *abstract states*. Then, we extend the semilattice’s partial order relation to abstract states $\sigma_1, \sigma_2 \in \text{MAP}$:

$$\sigma_1 \sqsubseteq \sigma_2 \iff \forall. x \in \text{VAR} . \sigma_1(x) \sqsubseteq \sigma_2(x)$$

We also extend the join operation to abstract states $\sigma_1, \sigma_2 \in \text{MAP}$:

$$(\sigma_1 \sqcup \sigma_2)(x) = \begin{cases} a \sqcup b & \text{if } \sigma_1(x) = a \text{ and } \sigma_2(x) = b \\ a & \text{if } \sigma_1(x) = a \text{ and } \sigma_2(x) \text{ is undefined} \\ b & \text{if } \sigma_1(x) \text{ is undefined and } \sigma_2(x) = b \\ \text{undefined} & \text{otherwise.} \end{cases}$$

The NPA’s flow function $\text{FLOW} : \text{INST}' \times \text{MAP} \rightarrow \text{MAP}$ is defined in Figure 7. Note, $\sigma_0 = \{x \mapsto \text{Null} : x \in \text{VAR}\}$. Also, we omit the `return y@a` case because it does not have CFG successors in a well-formed program.

4.2.1 Properties

It can be shown that this flow function is monotonic: for any $\iota \in \text{INST}'$ and abstract states $\sigma_1, \sigma_2 \in \text{MAP}$, if $\sigma_1 \sqsubseteq \sigma_2$ then $\text{FLOW}[\iota](\sigma_1) \sqsubseteq \text{FLOW}[\iota](\sigma_2)$. It can also be shown that the flow function is locally sound, *i.e.* the flow function models the concrete semantics at each

$$\begin{aligned}
\text{FLOW}(x := y, \sigma) &= \sigma[x \mapsto \sigma(y)] \\
\text{FLOW}(\text{branch } x, \sigma) &= \sigma \\
\text{FLOW}(\text{if } x, \sigma) &= \sigma[x \mapsto \text{NonNull}] \\
\text{FLOW}(\text{else } x, \sigma) &= \sigma[x \mapsto \text{Null}] \\
\text{FLOW}(x := m@a(y@b), \sigma) &= \sigma[x \mapsto a] \\
\text{FLOW}(\text{proc } m@a(y@b), \sigma) &= \sigma_0[y \mapsto b] \\
\text{FLOW}(x := \text{null}, \sigma) &= \sigma[x \mapsto \text{Null}] \\
\text{FLOW}(x := \text{new}(\bar{f}), \sigma) &= \sigma[x \mapsto \text{NonNull}] \\
\text{FLOW}(x := y \wedge z, \sigma) &= \begin{cases} \sigma[x \mapsto \text{Null}] & \text{if } \text{Null} \in \{\sigma(y), \sigma(z)\} \\ \sigma[x \mapsto \text{Nullable}] & \text{if } \text{Nullable} \in \{\sigma(y), \sigma(z)\} \\ \sigma[x \mapsto \text{NonNull}] & \text{otherwise} \end{cases} \\
\text{FLOW}(x := y \vee z, \sigma) &= \begin{cases} \sigma[x \mapsto \text{NonNull}] & \text{if } \text{NonNull} \in \{\sigma(y), \sigma(z)\} \\ \sigma[x \mapsto \text{Nullable}] & \text{if } \text{Nullable} \in \{\sigma(y), \sigma(z)\} \\ \sigma[x \mapsto \text{Null}] & \text{otherwise} \end{cases} \\
\text{FLOW}(x := y.f, \sigma) &= \sigma[x \mapsto \text{Nullable}][y \mapsto \text{NonNull}] \\
\text{FLOW}(x.f := y, \sigma) &= \sigma[x \mapsto \text{NonNull}] \\
\text{FLOW}(\text{main}, \sigma) &= \sigma_0
\end{aligned}$$

■ **Figure 7** All consequential cases of the flow function used by NPA.

step. To express this property formally, we define the predicate $\text{DESC}(\rho, \sigma)$ on $\text{ENV} \times \text{MAP}$, which says that the abstract state σ “describes” the concrete environment ρ :

$$\text{DESC}(\rho, \sigma) \iff \text{for all } x \in \text{VAR} . \rho(x) \in \text{CONC}(\sigma(x)).$$

Then, if $\langle S' \cdot \langle \rho, [l]_v \rangle \cdot S \parallel \mu \rangle \xrightarrow{p} \langle \langle \rho', v' \rangle \cdot S \parallel \mu' \rangle$, it must be the case that

$$\text{DESC}(\rho, \sigma) \implies \text{DESC}(\rho', \text{FLOW}[[l]](\sigma)) \text{ for all } \sigma \in \text{MAP}.$$

4.3 Fixpoint Algorithm

This brings us to Algorithm 1 [16], which is used to analyze a program and compute whether each program variable is `Nullable`, `NonNull`, or `Null` at each program point (the program results π). More specifically, the algorithm applies the flow function to each program instruction recording or updating the results until a fixpoint is reached – *i.e.* until the results stop changing (becoming more approximate). The algorithm will always reach a fixpoint (terminate), because `FLOW` is monotone and the height of the semilattice (Sec. 4.1) is finite. Note, the algorithm does not specify the order in which instructions are analyzed, because the order does not affect the results when `FLOW` is monotonic. An implementation may choose to analyze instructions in CFG order – following the directed edges of the CFG.

4.4 Safety Function & Static Warnings

Next, we present a way to use analysis results π produced by the fixpoint algorithm to determine whether to accept or reject a given program. Our goal is to ensure that when

■ **Algorithm 1** Kildall’s worklist algorithm.

```

1: function KILDALL( $\text{FLOW}, \sqcup, p$ )
2:    $\pi \leftarrow \{v \mapsto \emptyset : v \in \text{VERT}_p\}$ 
3:    $V \leftarrow \text{VERT}_p$   $\triangleright V \subseteq \text{VERT}_p$ 
4:   while  $V \neq \emptyset$  do
5:      $[\iota]_v \leftarrow$  an element of  $V$   $\triangleright v \in V$  and  $\iota = \text{INST}_p(v)$ 
6:      $V \leftarrow V \setminus \{v\}$   $\triangleright v \notin V$ 
7:      $\sigma \leftarrow \pi(v)$ 
8:      $\sigma' \leftarrow \text{FLOW}[[\iota]](\sigma)$ 
9:     for  $v \xrightarrow{p} u$  do  $\triangleright u \in \text{VERT}_p$ 
10:      if  $\sigma' \sqcup \pi(u) \neq \pi(u)$  then  $\triangleright$  think of as  $\sigma' \not\sqsubseteq \pi(u)$ 
11:         $\pi(u) \leftarrow \pi(u) \sqcup \sigma'$ 
12:         $V \leftarrow V \cup \{u\}$ 
13:      end if
14:    end for
15:  end while
16:  return  $\pi$ 
17: end function

```

```

SAFE( $x := m@a(y@b), y$ ) =  $b$ 
SAFE(return  $y@a$ ,  $y$ ) =  $a$ 
SAFE( $x := y.f$ ,  $y$ ) = NotNull
SAFE( $x.f := y$ ,  $x$ ) = NotNull

```

■ **Figure 8** All nontrivial cases of the safety function.

we run the program, it will not get stuck; that is, for any state ξ that the program reaches, we want to ensure that either ξ is a final state $\langle E \cdot \text{nil} \parallel \mu \rangle$ or there is another state ξ' such that $\xi \rightarrow_p \xi'$. To do this, we define the safety function $\text{SAFE}[[\iota]](x) : \text{INST}' \times \text{VAR} \rightarrow \text{ABST}$, which returns the abstract value representing the set of “safe” values x can take on before ι is executed. Figure 8 gives a few representative cases for SAFE , and in all the cases not shown SAFE returns **NotNullable**. In particular, a procedure call’s argument must adhere to the procedure’s parameter annotation, a return value must adhere to its corresponding return annotation, and all field accesses must have non-null receivers. Therefore, the safety function guards against all undefined behavior.

4.4.1 Static Warnings

Now, we can state the meaning of a valid program $p \in \text{PROG}'$:

for all $[\iota]_v \in \text{VERT}_p$ and $x \in \text{VAR}$. $\pi(v) = \sigma \implies \sigma(x) \sqsubseteq \text{SAFE}[[\iota]](x)$

where $\pi = \text{KILDALL}(\text{FLOW}, \sqcup, p)$.

That is, NPA emits static warnings when the fixpoint results disagree, according to the partial order \sqsubseteq , with the safety function. Also, we prove in Section 4.5 that a valid program does not get stuck.

4.5 Soundness of NPA

As discussed above, PICL's semantics are designed to get stuck when procedure annotations are violated or when null objects are dereferenced. Therefore, informally *soundness* says that a valid program does not get stuck during execution. Formally, soundness is defined with progress and preservation statements. Before their statement we must first define the notion of valid states to complement our definition of valid programs:

Let $p \in \text{PROG}'$. A state $\xi = \langle \langle \rho_1, v_1 \rangle \cdot \langle \rho_2, v_2 \rangle \cdots \langle \rho_n, v_n \rangle \cdot \text{nil} \parallel \mu \rangle \in \text{STATE}_p$ is *valid* if

for all $1 \leq i \leq n$. $\text{DESC}(\rho_i, \pi(v_i))$ where $\pi = \text{KILDALL}(\text{FLOW}, \sqcup, p)$.

A state is *valid* if it is described by the static analysis results π .

► **Proposition 1** (static progress). *Let $p \in \text{PROG}'$ be valid. If $\xi = \langle E_1 \cdot E_2 \cdot S \parallel \mu \rangle \in \text{STATE}_p$ is valid then $\xi \longrightarrow_p \xi'$ for some $\xi' \in \text{STATE}_p$.*

► **Proposition 2** (static preservation). *Let $p \in \text{PROG}'$ be valid. If $\xi \in \text{STATE}_p$ is valid and $\xi \longrightarrow_p \xi'$ then ξ' is valid.*

5 Gradual Null-Pointer Analysis

In this section, we derive GNPA from NPA, presented previously (Sec. 4). We proceed following the Abstracting Gradual Typing methodology introduced by Garcia *et al.* [14] in the context of gradual type systems, adapting it to fit the concepts of static analysis.

We present the GNPA's lifted semilattice (Sec. 5.1), flow and safety functions (Sec. 5.2), and fixpoint algorithm (Sec. 5.3). We also discuss how static (Sec. 5.4) and run-time warnings (Sec. 5.5) are generated by the analysis. Finally, Section 5.6 establishes the main properties of GNPA.

Note, here, annotations may be missing, so we extend our set of annotations with $?$: $\text{ANN} = \{\text{NonNull}, \text{Nullable}\} \cup \{?\}$.

5.1 Lifting the Semilattice

In this section, we lift the semilattice $(\text{ABST}, \sqsubseteq, \sqcup)$ (Sec. 4.1) by following the Abstracting Gradual Typing (AGT) framework [14]. First, we extend the set of semilattice elements ABST to the new set $\widetilde{\text{ABST}} \supseteq \text{ABST}$:

$$\widetilde{\text{ABST}} = \text{ABST} \cup \{?\} \cup \{a? : a \in \text{ABST}\} =$$

$$\{\text{Nullable}, \text{NonNull}, \text{Null}, ?, \text{NonNull?}, \text{Null?}\}.$$

Note that we equate the elements Nullable? and Nullable in $\widetilde{\text{ABST}}$. In Section 5.1.1, we give the semantics of the new lattice elements resulting in $\top = \text{Nullable?} = \text{Nullable}$. If ABST had a bottom element \perp , then $\perp = \perp?$ similarly.

The join \sqcup and partial order \sqsubseteq are also lifted to their respective counterparts $\widetilde{\sqcup}$ (Sec. 5.1.2) and $\widetilde{\sqsubseteq}$ (Sec. 5.1.3). The resulting lifted semilattice $(\widetilde{\text{ABST}}, \widetilde{\sqcup})$ with lifted relation $\widetilde{\sqsubseteq}$ underpins the optimism in GNPA.

5.1.1 Giving Meaning to Missing Annotations

A straightforward way to handle $?$ would be to make it the top element $? = \top$ or the bottom element $? = \perp$ of NPA's semilattice. However, neither choice is sufficient for our goal:

- If $? = \perp$, then $? \sqsubseteq a$ for all $a \in \text{ABST}$ and $\text{CONC}(\perp) = \emptyset$. As a result, if the return annotation of a procedure was $?$, then we could use the return value in any context without the analysis giving a warning. But, anytime an initialized variable is checked against the $?$ annotation, such as checking the non-null return value y against the $?$ return annotation $\text{NonNull} \sqsubseteq ?$, the check will fail as $a \not\sqsubseteq ?$ for all $a \in \text{ABST} . a \neq \perp$.
- If we let $? = \top$ then we have $a \sqsubseteq ?$ for all $a \in \text{ABST}$. Therefore, we can pass any argument to a parameter annotated as $?$ without the static part of GNPA giving a warning. But, if the return annotation of that procedure is $?$, then the analysis will produce false positives in caller contexts wherever the return value is dereferenced. In other words, our analysis would operate exactly as `PolyNull` for the example in Fig. 1, which is not ideal.

Our goal is to construct an analysis system that does not produce false positive static warnings when a developer omits an annotation. To achieve this, we draw on work in gradual typing [14]. We define the injective concretization function $\gamma : \widetilde{\text{ABST}} \rightarrow \mathcal{P}^+(\text{ABST})$ where $\widetilde{\text{ABST}} \supseteq \text{ABST}$ is the lifted semilattice element set (Sec. 5.1):

$$\gamma(a) = \{a\} \quad \text{for } a \in \text{ABST}, \quad \gamma(?) = \text{ABST}, \quad \text{and} \quad \gamma(a?) = \{b \in \text{ABST} : a \sqsubseteq b\}.$$

An element in ABST is mapped to itself as it can only represent itself. In contrast, $?$ may represent any element in ABST at all times to support optimism in all possible contexts. Further, $a?$ means “ a or anything more general than it,” in contrast to a gradual formula $\phi \wedge ?$ that means “ ϕ or anything more specific than it” [2]. As a result, $a?$ does not play the intuitive role of “supplying missing information,” as it would in gradual verification. Instead, $a?$ is simply an artifact of our construction, which is why the only element of $\text{ANN} \setminus \text{ABST}$ is $?$.

Then, if $\gamma(\tilde{a}) \subseteq \gamma(\tilde{b})$ for some $\tilde{a}, \tilde{b} \in \widetilde{\text{ABST}}$, we write $\tilde{a} \lesssim \tilde{b}$ and say that \tilde{a} is more precise than \tilde{b} . Further, $\iota_1 \lesssim \iota_2$ means that 1) the two instructions are equal except for their annotations, and 2) the annotations in ι_1 are more precise than the corresponding annotations in ι_2 .

5.1.2 Lifted Join $\tilde{\sqcup}$

We begin by introducing a semilattice definition [9], which states that a semilattice is an algebraic structure (S, \sqcup) where for all $x, y, z \in S$ the following hold:

- $x \sqcup (y \sqcup z) = (x \sqcup y) \sqcup z$ (associativity)
- $x \sqcup y = y \sqcup x$ (commutativity)
- $x \sqcup x = x$ (idempotency)

Then, we write $x \sqsubseteq y$ when $x \sqcup y = y$ and it can be shown this \sqsubseteq is a partial order. Recall that NPA uses \sqcup in Algorithm 1 to compute a fixpoint that describes the behavior of a program p . The fixpoint can only be reached when \sqcup is idempotent. Similarly, \sqcup must be commutative and associative so that program instructions can be analyzed in any order. Thus, our extended join operation $\tilde{\sqcup} : \widetilde{\text{ABST}} \times \widetilde{\text{ABST}} \rightarrow \widetilde{\text{ABST}}$ must be associative, commutative, and idempotent making $(\widetilde{\text{ABST}}, \tilde{\sqcup})$ a join-semilattice.

3:14 Gradual Program Analysis for Null Pointers

To define such a function we turn to insights from gradual typing [14]. We define an abstraction function $\alpha : \mathcal{P}^+(\text{ABST}) \rightarrow \widetilde{\text{ABST}}$, which forms a Galois connection with γ :

$$\alpha(\widehat{a}) = \gamma^{-1} \left(\bigcap_{\substack{\tilde{b} \in \widetilde{\text{ABST}} \\ \gamma(\tilde{b}) \supseteq \widehat{a}}} \gamma(\tilde{b}) \right)$$

where, for $a \in \text{ABST}$, γ^{-1} is:

$$\gamma^{-1}(\{a\}) = a \quad \gamma^{-1}(\text{ABST}) = ? \quad \gamma^{-1}(\{b \in \text{ABST} : a \sqsubseteq b\}) = a?.$$

Then we define the join of $\tilde{a}, \tilde{b} \in \widetilde{\text{ABST}}$ as follows:

$$\tilde{a} \sqcup \tilde{b} = \alpha(\{a \sqcup b : a \in \gamma(\tilde{a}) \text{ and } b \in \gamma(\tilde{b})\})$$

For example,

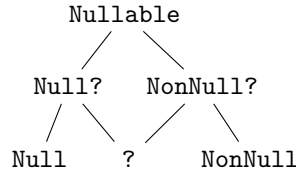
$$\begin{aligned} \text{NonNull} \sqcup ? &= \alpha(\{a \sqcup b : a \in \{\text{NonNull}\} \text{ and } b \in \text{ABST}\}) & (1) \\ &= \alpha(\{\text{NonNull}, \text{Nullable}\}) & (2) \\ &= \gamma^{-1}(\gamma(\text{NonNull?}) \cap \gamma(?)) & (3) \\ &= \gamma^{-1}(\{\text{NonNull}, \text{Nullable}\} \cap \text{ABST}) & (4) \\ &= \gamma^{-1}(\{\text{NonNull}, \text{Nullable}\}) & (5) \\ &= \text{NonNull?} & (6) \end{aligned}$$

That is, the join of all the ABST elements represented by `NonNull` and `?` results in the set `{NonNull, Nullable}` (1, 2). Applying α to this set is equivalent to applying γ^{-1} to $\gamma(\text{NonNull?}) \cap \gamma(?)$ (3); because, the only $\widetilde{\text{ABST}}$ elements that represent both `NonNull` and `Nullable` are `NonNull?` and `?`. The intersection of $\gamma(\text{NonNull?})$ and $\gamma(?)$ is `{NonNull, Nullable}` (4, 5), so we are really applying γ^{-1} to `{NonNull, Nullable}` (5). Therefore, $\text{NonNull} \sqcup ? = \text{NonNull?}$ (6). Notice, the intersection of the representative sets $\gamma(\text{NonNull?})$ and $\gamma(?)$ of `{NonNull, Nullable}` = \widehat{a} is used to find the most precise element in $\widetilde{\text{ABST}}$ that can represent \widehat{a} .

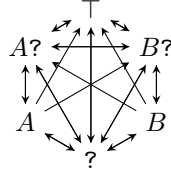
Now we return to the properties of \sqcup . Since \sqcup is commutative, we have that \sqcup is commutative. Idempotency is also not too onerous: it is equivalent to the condition that every element of $\widetilde{\text{ABST}}$ represents a subsemilattice of ABST. That is, for every $\tilde{a} \in \widetilde{\text{ABST}}$ and $a_1, a_2 \in \gamma(\tilde{a})$, we must have $a_1 \sqcup a_2 \in \gamma(\tilde{a})$. This is true by construction. Associativity is tricky and motivates our complex definition of $\widetilde{\text{ABST}}$. Ideally, $\widetilde{\text{ABST}}$ would be defined simply as $\text{ABST} \cup \{?\}$, however in this case \sqcup is not associative:

$$\begin{aligned} \text{Null} \sqcup (\text{NonNull} \sqcup ?) &= \text{Null} \sqcup ? \\ &= ? \\ &\neq \text{Nullable} \\ &= \text{Nullable} \sqcup ? \\ &= (\text{Null} \sqcup \text{NonNull}) \sqcup ?. \end{aligned}$$

Fortunately, our definition of $\widetilde{\text{ABST}}$ which also includes the intermediate optimistic elements `NonNull?` and `Null?` results in an associative \sqcup function and a finite-height semilattice ($\widetilde{\text{ABST}}, \sqcup$). Figure 9 shows the semilattice structure induced by \sqcup .



■ **Figure 9** The semilattice structure induced by the lifted join $\tilde{\sqcup}$. Specifically, this is the Hasse diagram of the partial order $\{(\tilde{a}, \tilde{b}) : \tilde{a} \tilde{\sqcup} \tilde{b} = \tilde{b}\}$.



■ **Figure 10** The lifted partial order, where each directed edge $\tilde{a} \rightarrow \tilde{b}$ means $\tilde{a} \tilde{\sqsubseteq} \tilde{b}$. (Self-loops are omitted). Here, `Nullable` is abbreviated \top , and `Null` and `NonNull` are abbreviated A and B respectively.

5.1.3 Lifted Order $\tilde{\sqsubseteq}$

Now it is fairly straightforward to construct $\tilde{\sqsubseteq}$. Recall, NPA emits static warnings when the fixpoint results disagree with the safety function, according to the partial order \sqsubseteq . The fixpoint results and the safety function now return elements in \widetilde{ABST} , so we lift \sqsubseteq to $\tilde{\sqsubseteq} \subseteq \widetilde{ABST} \times \widetilde{ABST}$ using the concretization function γ :

$$\tilde{a} \tilde{\sqsubseteq} \tilde{b} \iff \exists . a \in \gamma(\tilde{a}) \text{ and } b \in \gamma(\tilde{b}) \text{ such that } a \sqsubseteq b \text{ for } \tilde{a}, \tilde{b} \in \widetilde{ABST}.$$

Figure 10 gives the lifted order relation $\tilde{\sqsubseteq}$ in graphical form.

The $\tilde{\sqsubseteq}$ predicate is a maximally permissive version of the \sqsubseteq predicate for `NonNull?`, `Null?`, and `?`. For example, $? \tilde{\sqsubseteq} \text{NonNull}$ since $\gamma(?) = \{\text{NonNull}, \text{Null}, \text{Nullable}\}$, $\gamma(\text{NonNull}) = \{\text{NonNull}\}$, and $\text{NonNull} \sqsubseteq \text{NonNull}$. By similar reasoning, $\text{NonNull} \tilde{\sqsubseteq} ?$. In fact, $? \tilde{\sqsubseteq} a \tilde{\sqsubseteq} ?$, $\text{NonNull?} \tilde{\sqsubseteq} a \tilde{\sqsubseteq} \text{NonNull?}$, and $\text{Null?} \tilde{\sqsubseteq} a \tilde{\sqsubseteq} \text{Null?}$ for $a \in ABST$. So, clearly $\tilde{\sqsubseteq}$ is not a partial order. The $\tilde{\sqsubseteq}$ predicate must be maximally permissive to support the optimism used in the `safeReverse` example from Figure 1 (Sec. 2.2): calls to `safeReverse` with null and non-null arguments are valid and dereferences of its return values are also valid. However, $\tilde{\sqsubseteq}$ is the same as \sqsubseteq when both of its arguments come from $ABST$, e.g. $\text{NonNull} \tilde{\sqsubseteq} \text{Nullable}$ and $\text{Nullable} \not\tilde{\sqsubseteq} \text{NonNull}$. This allows our gradual analysis to apply NPA where annotations are complete enough to support it.

5.1.4 Properties

We previously mentioned some of the properties which $(\widetilde{ABST}, \tilde{\sqcup})$ satisfy. Here, we formally state them, and their proofs can be found in the appendix of the full version of this paper [10].

▶ **Proposition 3.** $(\widetilde{ABST}, \tilde{\sqcup})$ is a semilattice; in other words, $\tilde{\sqcup}$ is associative, idempotent, and commutative.

▶ **Proposition 4.** If the height of $(ABST, \sqcup)$ is $n > 0$, then the height of $(\widetilde{ABST}, \tilde{\sqcup})$ is $n + 1$ (in particular, $(\widetilde{ABST}, \tilde{\sqcup})$ has finite height).

5.2 Lifting the Flow & Safety Functions

Now both instructions and abstract states ($\tilde{\sigma} \in \widetilde{\text{MAP}} = \text{VAR} \rightarrow \widetilde{\text{ABST}}$) may contain optimistic abstract values. Therefore, similar to lifting the join \sqcup , we follow the AGT *consistent function lifting* approach [14] when defining GNPA's flow function $\widetilde{\text{FLOW}} : \text{INST} \times \widetilde{\text{MAP}} \rightarrow \widetilde{\text{MAP}}$ for this new domain.

Specifically, for $\iota \in \text{INST}$ and $\tilde{\sigma} = \{x \mapsto \tilde{a}_x : x \in \text{VAR}\} \in \widetilde{\text{MAP}}$, we define

$$\begin{aligned} \widetilde{\text{FLOW}}[z := m@a(y@b)](\tilde{\sigma}) &= \{x \mapsto \alpha(\{(\text{FLOW}[z := m@a'(y@b')](\sigma'))(x) \\ &\quad : a' \in \gamma(a) \wedge b' \in \gamma(b) \wedge \sigma' \in \Sigma\}) : x \in \text{VAR}\} \\ \widetilde{\text{FLOW}}[\text{proc } m@a(y@b)](\tilde{\sigma}) &= \{x \mapsto \alpha(\{(\text{FLOW}[\text{proc } m@a'(y@b')](\sigma'))(x) \\ &\quad : a' \in \gamma(a) \wedge b' \in \gamma(b) \wedge \sigma' \in \Sigma\}) : x \in \text{VAR}\} \\ \widetilde{\text{FLOW}}[\iota](\tilde{\sigma}) &= \{x \mapsto \alpha(\{(\text{FLOW}[\iota](\sigma'))(x) : \sigma' \in \Sigma\}) : x \in \text{VAR}\} \quad \text{otherwise} \end{aligned}$$

where $\Sigma = \{\{x \mapsto a_x : x \in \text{VAR}\} : a_x \in \gamma(\tilde{a}_x) \text{ for all } x \in \text{VAR}\}$.

Note that the procedure call and procedure entry instructions are the only instructions in $\widetilde{\text{FLOW}}$'s domain that may contain ? annotations, so the corresponding $\widetilde{\text{FLOW}}$ rules are lifted with respect to those annotations. Similarly, all rules are lifted with respect to their abstract states.

Recall that we defined the predicate DESC on $\text{ENV} \times \text{MAP}$ to express the local soundness of FLOW . For $\widetilde{\text{FLOW}}$, we lift DESC to $\widetilde{\text{DESC}}$ on $\text{ENV} \times \widetilde{\text{MAP}}$ such that it is maximally permissive like the $\widetilde{\sqsubseteq}$ predicate:

$$\widetilde{\text{DESC}}(\rho, \tilde{\sigma}) \iff \text{DESC}(\rho, \sigma) \text{ for some } \sigma \in \Sigma$$

where Σ is constructed in the same way as for $\widetilde{\text{FLOW}}$.

Finally, we again follow the consistent function lifting methodology to construct $\widetilde{\text{SAFE}} : \text{INST} \times \text{VAR} \rightarrow \widetilde{\text{ABST}}$ from $\text{SAFE} : \text{INST}' \times \text{VAR} \rightarrow \text{ABST}$:

$$\begin{aligned} \widetilde{\text{SAFE}}[z := m@a(y@b)](x) &= \alpha(\{\text{SAFE}[z := m@a'(y@b')](x) : a' \in \gamma(a) \wedge b' \in \gamma(b)\}) \\ \widetilde{\text{SAFE}}[\text{proc } m@a(y@b)](x) &= \alpha(\{\text{SAFE}[\text{proc } m@a'(y@b')](x) : a' \in \gamma(a) \wedge b' \in \gamma(b)\}) \\ \widetilde{\text{SAFE}}[\text{return } y@a](x) &= \alpha(\{\text{SAFE}[\text{return } y@a'](x) : a' \in \gamma(a)\}) \\ \widetilde{\text{SAFE}}[\iota](x) &= \alpha(\text{SAFE}[\iota](x)) \quad \text{otherwise} \end{aligned}$$

Other than the casewise-defined FLOW rules for \wedge and \vee , the lifted $\widetilde{\text{FLOW}}$ and $\widetilde{\text{SAFE}}$ functions simplify down to the same computation rules as FLOW and SAFE as shown in Figure 7 and Figure 8 respectively, replacing FLOW with $\widetilde{\text{FLOW}}$ and SAFE with $\widetilde{\text{SAFE}}$.

5.3 Lifting the Fixpoint Algorithm

To lift the fixpoint algorithm, we simply plug $\widetilde{\text{FLOW}}$ and $\widetilde{\sqcup}$ into Algorithm 1 to compute $\tilde{\pi} = \text{KILDALL}(\widetilde{\text{FLOW}}, \widetilde{\sqcup}, p) : \text{VERT}_p \rightarrow \widetilde{\text{MAP}}$ for any $p \in \text{PROG}$.

5.4 Static Warnings

Using the lifted safety function, we say that a partially-annotated program $p \in \text{PROG}$ is *statically valid* if

$$\text{for all } [\iota]_v \in \text{VERT}_p \text{ and } x \in \text{VAR}, \quad \tilde{\pi}(v) = \tilde{\sigma} \implies \tilde{\sigma}(x) \widetilde{\sqsubseteq} \widetilde{\text{SAFE}}[\iota](x)$$

where $\tilde{\pi} = \text{KILDALL}(\widetilde{\text{FLOW}}, \tilde{\sqcup}, p)$.

Each piece of GNPA's static system ($(\widetilde{\text{ABST}}, \tilde{\sqcup})$, $\widetilde{\text{CONC}}$, $\widetilde{\text{FLOW}}$, $\widetilde{\text{SAFE}}$, and the fixpoint algorithm) is designed to be maximally optimistic for missing annotations. Therefore, the resulting system will not produce false positive warnings due to missing annotations. The system is also designed to apply NPA where annotations are available to support it, so it will still warn about violations of procedure annotations or null object dereferences where possible. See Section 2.2 for more information.

5.5 Dynamic Checking

GNPA's static system reduces false positive warnings at the cost of soundness. For example, as in Section 2.3, the analysis may assume a variable with a ? annotation is non-null to satisfy an object dereference when the variable is actually null. In order to avoid false negatives and ensure that our gradual analysis is sound, we modify the semantics of PICL to insert run-time checks where the analysis may be unsound. That is, if p is *statically valid* and there are program points $[l]_v$ such that

$$a \not\sqsubseteq \bigsqcup \gamma(\widetilde{\text{SAFE}}[[l]](x)) \quad \text{for some } x \in \text{VAR} \quad \text{and} \quad a \in \gamma((\tilde{\pi}(v))(x)),$$

then a run-time check must be inserted at those points to ensure the value of x is in $\text{CONC}(\bigsqcup \gamma(\widetilde{\text{SAFE}}[[l]](x)))$.

More precisely, we define a dedicated error state **error** and expand the set of run-time states to be $\widetilde{\text{STATE}}_p = \text{STATE}_p \cup \{\mathbf{error}\}$. Then we define a restricted semantics \xrightarrow{p} on $\widetilde{\text{STATE}}_p \times \widetilde{\text{STATE}}_p$ as follows. Let $\xi \in \text{STATE}_p$. If

$$\xi = \langle \langle \rho, [l] \rangle \cdot S \parallel \mu \rangle \quad \text{and} \quad \neg \widetilde{\text{DESC}}(\rho, \{x \mapsto \widetilde{\text{SAFE}}[[l]](x) : x \in \text{VAR}\})$$

then $\xi \xrightarrow{p} \mathbf{error}$. If there is some $\xi' \in \text{STATE}_p$ such that $\xi \rightarrow_p \xi'$, then $\xi \xrightarrow{p} \xi'$. Otherwise, there is no $\xi' \in \widetilde{\text{STATE}}_p$ such that $\xi \xrightarrow{p} \xi'$.

5.6 Gradual Properties

GNPA is sound, *conservative extension* of NPA – the static system is applied in full to programs with complete annotations, and adheres to the gradual guarantees inspired by Siek *et al.* [23]. The gradual guarantees ensure losing precision is harmless, *i.e.* increasing the number of missing annotations in a program does not break its validity or reducibility.

To formally present each property, we first extend the notion of a valid state. Let $p \in \text{PROG}$. A state $\xi = \langle \langle \rho_1, v_1 \rangle \cdot \langle \rho_2, v_2 \rangle \cdots \langle \rho_n, v_n \rangle \cdot \text{nil} \parallel \mu \rangle \in \text{STATE}_p$ is valid if

$$\text{for all } 1 \leq i \leq n, \quad \widetilde{\text{DESC}}(\rho_i, \tilde{\pi}(v_i)) \quad \text{where} \quad \tilde{\pi} = \text{KILDALL}(\widetilde{\text{FLOW}}, \tilde{\sqcup}, p).$$

Then, for fully-annotated programs, GNPA and the modified semantics are conservative extensions of NPA and PICL's semantics, respectively.

► **Proposition 5** (conservative static extension).

If $p \in \text{PROG}'$ then $\text{KILDALL}(\text{FLOW}, \sqcup, p) = \text{KILDALL}(\widetilde{\text{FLOW}}, \tilde{\sqcup}, p)$.

► **Proposition 6** (conservative dynamic extension). Let $p \in \text{PROG}'$ be valid, and let $\xi_1, \xi_2 \in \text{STATE}_p$. If ξ_1 is valid then $\xi_1 \rightarrow_p \xi_2$ if and only if $\xi_1 \xrightarrow{p} \xi_2$.

GNPA is sound, *i.e.* valid programs will not get stuck during execution. However, programs may step to a dedicated **error** state when run-time checks fail. Soundness is stated with a progress and preservation argument.

► **Proposition 7** (gradual progress). *Let $p \in \text{PROG}$ be valid. If $\xi = \langle E_1 \cdot E_2 \cdot S \parallel \mu \rangle \in \text{STATE}_p$ is valid then $\xi \xrightarrow{p} \tilde{\xi}'$ for some $\tilde{\xi}' \in \widetilde{\text{STATE}}_p$.*

► **Proposition 8** (gradual preservation). *Let $p \in \text{PROG}$ be valid. If $\xi \in \text{STATE}_p$ is valid and $\xi \xrightarrow{p} \xi'$ for some $\xi' \in \text{STATE}_p$, then ξ' is valid.*

Finally, GNPA satisfies both the static and dynamic gradual guarantees. Both of the guarantees rely on a definition of *program precision*. Specifically, if programs p_1 and p_2 are identical except perhaps that some annotations in p_2 are $?$ where they are not $?$ in p_1 , then we say that p_1 is more precise than p_2 , and write $p_1 \lesssim p_2$.

Then, the *static gradual guarantee* states that increasing the number of missing annotations in a valid program does not introduce static warnings (*i.e.* break program validity).

► **Proposition 9** (static gradual guarantee). *Let $p_1, p_2 \in \text{PROG}$ such that $p_1 \lesssim p_2$. If p_1 is statically valid then p_2 is statically valid.*

The *dynamic gradual guarantee* ensures that increasing the number of missing annotations in a program does not change the observable behavior of the program (*i.e.* break program reducibility for valid programs).

► **Proposition 10** (dynamic gradual guarantee). *Let $p_1, p_2 \in \text{PROG}$ be statically valid, where $p_1 \lesssim p_2$. Let $\xi_1, \xi_2 \in \text{STATE}_{p_2}$. If $\xi_1 \xrightarrow{p_1} \xi_2$ then $\xi_1 \xrightarrow{p_2} \xi_2$.*

Note, the small-step semantics $\xrightarrow{\cdot}$ are designed to make the proofs of the aforementioned properties easier at the cost of easily implementable run-time checks. Therefore, we give the following proposition that connects a more implementable design to $\xrightarrow{\cdot}$. That is, we can use the contrapositive of this proposition to implement more optimal run-time checks. Specifically, the naïve implementation would check each variable at each program point to make sure it satisfies the safety function for the instruction about to be executed. But Proposition 1 tells us that we only need to check variables at runtime when our analysis results don't already guarantee (statically) that they will satisfy the safety function.

► **Proposition 11** (run-time checks). *Let $p \in \text{PROG}$ be valid according to $\tilde{\pi} = \text{KILDALL}(\widetilde{\text{FLOW}}, \tilde{\sqcup}, p)$, and let $\xi = \langle \langle \rho, [l]_v \rangle \cdot S \parallel \mu \rangle \in \text{STATE}_p$ be valid. If $\xi \xrightarrow{p} \text{error}$ then there is some $x \in \text{VAR}$ and $a \in \gamma((\tilde{\pi}(v))(x))$ such that $a \not\sqsubseteq \sqcup \gamma(\widetilde{\text{SAFE}}[l](x))$.*

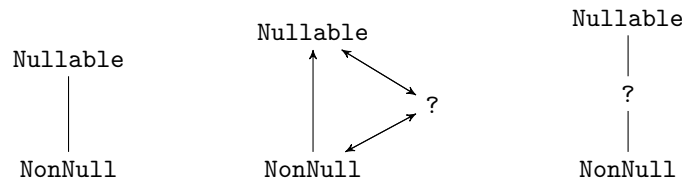
6 Preliminary Empirical Evaluation

In this section, we discuss the implementation of GNPA and two studies designed to evaluate its usefulness in practice. Preliminary evidence suggests that our analysis can be used at scale, produces fewer false positives than state-of-the-art tools, and eliminates on average more than half of the null-pointer checks Java automatically inserts at run time. These results illustrate an important practical difference between GNPA and other null-pointer analyses. While a sound static analysis can be used to prove the redundancy of run-time checks, and an unsound static analysis can be used to reduce the number of false positives, neither of those can do both at the same time. On the other hand, GNPA can both prove the redundancy of run-time checks and reduce reported false positives

6.1 Research Questions

We seek answers to the following questions:

1. Can a gradual null-pointer analysis be effectively implemented and used at scale?



■ **Figure 11** *Left*: The starting null-pointer semilattice for Graduator. *Middle*: The lifted partial ordering, where each directed edge $\tilde{a} \rightarrow \tilde{b}$ means $\tilde{a} \sqsubseteq \tilde{b}$. (Self-loops are omitted.) *Right*: The semilattice structure induced by the lifted join $\tilde{\sqcup}$.

2. Does such a null-pointer analysis produce fewer false positives than industry-grade analyses?
3. Does the gradual null-pointer analysis perform significantly fewer null-pointer checks than the naïve approach of checking every dereference?

6.2 Prototype

Facebook Infer provides a framework to construct static analyses that use abstract interpretation. We built a prototype of GNPA, called *Graduator*, in this framework. Our prototype uses Infer’s HIL intermediate language representation (IR). As a result, Graduator can be used to analyze code written in C, C++, Objective-C, and Java.

The preceding case study (Secs. 3–5) uses a base semilattice with three elements, `Null`, `Nonnull`, and `Nullable`, in order to demonstrate that a semilattice lifting may contain additional intermediate optimistic elements, `Null?` and `Nonnull?`. For simplicity, we implemented the semilattice from Figure 11, along with its lifted variant, order relation and join function, in our prototype. This semilattice is the same as the base one in the case study except it does not contain `Null`: the initial static semilattice has only `Nonnull` and `Nullable`, and the gradual semilattice only adds one additional `?` element. There are a couple other differences between our formalism and our Graduator prototype, one of which is that Graduator allows field annotations while our formalism does not.

Infer does not support modifying Java source code, so Graduator simply reports the locations where it should insert run-time checks rather than inserting them directly. In fact, Graduator may output any of the following:

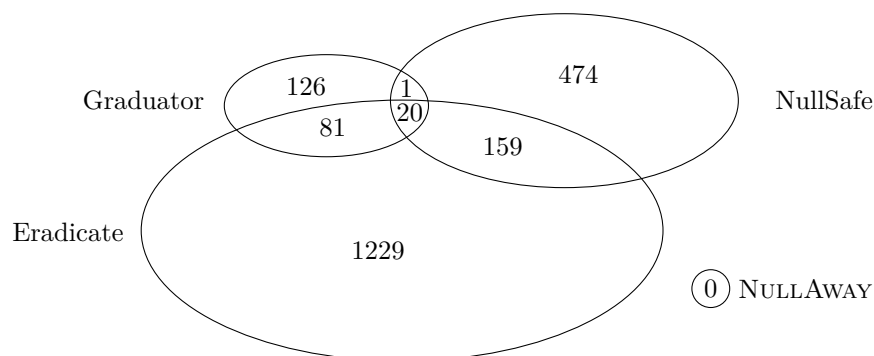
- `GRADUAL_STATIC` – a static warning.
- `GRADUAL_CHECK` – a location to check a possibly-null dereference.
- `GRADUAL_BOUNDARY` – another location to insert a check, such as passing an argument to a method, returning from a method, or assigning a value to a field.

Since Java checks for null-pointer dereferences automatically, soundness is preserved. A more complete implementation of GNPA would insert run-time checks as part of the build process. As a result, some bugs may be caught earlier when the gradual analysis inserts checks at method boundaries and field assignments.

By implementing Graduator with Infer’s framework, Graduator is guaranteed to operate at scale. We also evaluate Graduator on a number of open source repositories as discussed in Sections 6.3 and 6.4. Thus, the answer to RQ1 is yes.

6.3 Static Warnings

To evaluate Graduator, we ran it on 15 of the 18 open-source Java repositories used to evaluate `NULLAWAY` [3] (we excluded 3 of the 18 repositories because we were unable to



■ **Figure 12** The total number of static warnings reported by the three Infer null checkers, for all 15 repositories.

successfully run Infer on them). We also ran NULLAWAY, and Infer’s existing null-pointer checkers Eradicate and NullSafe, on the repositories. Figure 12 shows the number of *static* warnings produced by each of these three checkers: 1489 for Eradicate, 654 for NullSafe, 228 for Graduator, and 0 for NULLAWAY, for a total of 2371.

Based on the NULLAWAY paper (in which Uber states that in practice they have found no instances of null-pointer dereferences caused by their tool’s unsoundness), it seems reasonable to assume that these repositories do not have null-pointer bugs, since NULLAWAY itself reports no static warnings for these repositories. After examining all 2371 warnings ourselves, we found that all but 57 (50 from Eradicate only, 2 from Graduator only, and 5 from Eradicate and Graduator but not NullSafe) were false positives due to systematic imprecision in the analysis tools. We were unable to determine whether the remaining 57 warnings represent actual bugs or not.

Under this assumption, Graduator reports significantly fewer false positives than Infer’s existing null-pointer checkers (although in this respect, it is of course outperformed by NULLAWAY) (RQ2). An interesting aspect of Figure 12 is how many warnings are produced by only one of the checkers: 1229 for Eradicate, 474 for NullSafe, and 126 for Graduator. Many of these warnings arose from generated and test case code.

6.3.1 Generated Code

Several of the 15 repositories generate code as part of their build process, and in some cases, the analysis tools gave warnings about the generated code. This accounts for

- 380 of the warnings given by NullSafe alone,
- 356 of the warnings given by Eradicate alone,
- 130 of the warnings given by both Eradicate and NullSafe but not Graduator, and
- 8 of the warnings given by Graduator alone.

Graduator reports significantly fewer static warnings for generated code, because such code is typically unannotated and Graduator is designed to be optimistic when annotations are missing.

6.3.2 Test Code

It is reasonable to assume that test code does not contain null dereference bugs, because if it did, then those bugs would show up when the tests are run. Static warnings about test code account for

■ **Table 1** Percentage of null-dereference checks which Graduator found to be redundant.

repository	dereference sites	eliminated checks	percent eliminated
keyvaluestore	419	156	37%
uLeak	620	241	39%
butterknife	2773	1129	41%
jib	5896	2499	42%
skaffold-tools-for-java	366	185	51%
picasso	2719	1458	54%
meal-planner	858	475	55%
caffeine	9455	5701	60%
AutoDispose	3218	1993	62%
ColdSnap	6360	4325	68%
ReactiveNetwork	2097	1626	78%
okbuck	19089	15130	79%
FloatingActionButtonSpeedDial	3049	2581	85%
QRContact	1272	1171	92%
OANDAFX	2216	2056	93%
overall	60407	40726	67%

- 384 of the warnings given by Eradicate alone, and
- 73 of the warnings given by both Eradicate and Graduator, but not NullSafe.

That is, Graduator reports fewer warnings for test code than Eradicate, but more than NullSafe. The NullSafe checker does not appear to treat test code specially, so it is unclear why NullSafe is performing better than Graduator for such code.

6.3.3 Remaining False Positives

The reader may wonder why Graduator reports any false positives on this codebase, since it intuitively seems that the static portion of a gradual analysis ought to be optimistic. Examining the warnings given by Graduator, we see that none of the warnings are due to treating missing annotations pessimistically; instead, they are due to places where the analysis has whatever annotations it needs, but the analysis is imprecise in other respects. For example, one common source of false positives is when a field is checked for null, then is read again. Our original static analysis is limited in that it does not treat fields flow-sensitively, causing false positives that are independent of the choice to be gradual or not with respect to annotations.

NULLAWAY avoids giving false positives on this same codebase, due to a combination of some unsound assumptions and a more precise analysis approach. While our approach for deriving gradual program analyses focuses on retaining soundness through a combination of static and dynamic checks, incorporating more precise analysis techniques (e.g. a flow-sensitive treatment of fields, perhaps in combination with a gradual alias analysis) could eliminate more of these false positives. In the meantime, our comparison to Eradicate and NullSafe is appropriate as these are the static analysis tools taking the most similar approach.

6.4 Run-time Checks

For the same set of 15 repositories analyzed by NULLAWAY, we performed another experiment using our prototype. We configured Graduator to ignore *all* annotations, so in effect, every field, argument, and return value was annotated as ?. For each repository, we counted all the locations where Graduator gave a GRADUAL_STATIC, GRADUAL_CHECK, or GRADUAL_BOUNDARY

warning, and compared that number to the total number of pointer dereferences in the code. By ignoring annotations, we ensured that each of these warnings appeared on dereferences, rather than allowing early checks at, e.g., method boundaries. We also ran analogous experiments with annotations enabled, but the number of run-time check warnings found were very similar to the numbers found with annotations disabled.

Table 1 shows what percentage of these dereference sites received no static warnings or run-time checks. Recall that Java automatically checks all dereferences to ensure that they are not null. Because GNPA is sound, this figure shows the percentage of null checks that are provably redundant, and could be safely removed by an ahead-of-time compiler.

Since we were able to eliminate an average of 67% of the null checks which Java automatically inserts, this experiment suggests the answer to RQ3 is yes. Note that these numbers only discuss the number of dereferences that appear in the code, and do not take into account which of these dereferences are executed more or less frequently at run-time.

This also illustrates an important practical difference between GNPA and other null-pointer analyses. While a sound static analysis can be used to prove the redundancy of run-time checks, and an unsound static analysis can be used to reduce the number of false positives, neither of those can do both at the same time. On the other hand, a gradual analysis can both prove the redundancy of run-time checks and reduce reported false positives.

7 Related Work

As discussed previously, our work builds on prior research in gradual typing: the criteria for gradual type systems [23] and the Abstracting Gradual Typing methodology, which develops a gradual type system from a purely static one [14]. In contrast to prior work in gradual typing, we address the challenges of tracking transitive dataflow relationships, rather than the local checks of typical type systems. In doing so, we gradualize, for the first time, the abstract interpretation of a program [8], and the canonical dataflow analysis fix-point algorithm [16].

The most closely related work in program analysis consists of *hybrid analyses*, which combine static and dynamic analysis techniques to counteract the weaknesses inherent to each approach. For example, Choi *et al.* [7] used a static analysis to substantially lower the run-time overhead of a dynamic data race analysis. Prior work on hybrid program analyses combines static and dynamic techniques in ad-hoc ways. Instead, we propose a principled methodology for deriving a hybrid (gradual) analysis from a static one, and show that the resulting analysis adheres to desirable properties such as soundness and the gradual guarantee.

There is a large body of literature on static program analysis, including multiple specialized conferences. Our work opens the door to gradual versions of them. Previously, we discussed existing null-pointer analysis tools [11], [3] and frameworks [21], and how GNPA is an improvement over them. Notably, our prototype is implemented in Infer’s framework [11].

The Granular type system [5] and the Blame for Null calculus [19] are gradual type systems for nullness, and thus solve a related problem to GNPA. The main difference in our work is that we use dataflow analysis instead of typing. This results in a significantly different user experience, as a full static specification within a gradual type system typically requires many more types to be specified (e.g. on all local variables) compared to a dataflow analysis, where for example we do not require (or even allow) nullity annotations on local variables. Basing our work on dataflow analysis also has a major impact on the technical development, requiring the novel lattice-based gradualization framework described in this

paper rather than the well-known type-based gradualization approaches used in Granular and Blame for Null. Blame for Null also investigates the notion of blame, which we leave for future work in the program analysis setting.

Contract checking [18, 13] can be used to check properties like nullness. Building on the idea of hybrid type checking [17], Xu *et al.* [25] explored how to check contracts using a hybrid of static and dynamic analysis. Their work was specialized to the context of logical assertions, whereas we are in the area of lattice-based program analyses. It is also unclear whether their approach conforms to the gradual guarantee.

O’Hearn *et al.* [20] proposed Incorrectness Logic as a means of proving that a program has a bug, rather than proving it correct. This is consistent with our goal of reducing false positives, but it stays in the realm of static reasoning, and therefore gives up soundness. In contrast, we reduce false positives without giving up soundness by adding run-time checks.

8 Conclusion

This paper is the first work on gradual program analysis. We introduced a framework which transforms abstract interpretation based static analyses relying on annotations into gradual ones. Gradual analyses handle missing annotations specially, allowing them to smoothly leverage both static and dynamic techniques. Static information is used where possible and dynamic information where necessary to reduce false positives while preserving soundness. Such analyses are also *conservative extensions* of their underlying static analyses and adhere to *gradual guarantees*, which state that losing precision is harmless. When presenting our framework, we developed a gradual null-pointer analysis, GNPA, with the previously mentioned properties that reduces false positives compared to some popular existing tools.

Importantly, the gradual framework can be applied as described to any abstract interpretation based static analysis under the following restrictions. The analysis should support annotations, have a finite-height semilattice, a monotonic, locally-sound flow function, a safety function, and operate on a first-order, procedural, imperative programming language. Additionally, checking membership in the semilattice should be decidable. Thus, initial followup work could include gradual taint analysis, to which our framework immediately applies. Finally, we do not support widening, but we do support context-sensitivity. In the future, we plan to explore extensions of our framework for infinite-height semilattices and widening; this would allow gradualization of other analyses, such as interval analysis. Still further work could include, for instance, pointer analyses, which do not have analogues in the field of gradual typing.

On the empirical side, there are further research questions to be answered: How often does a gradual analysis catch bugs statically versus how often does it catch them at run time? Is performance lost or gained when run time checks are inserted earlier via annotations rather than just-in-time? Finally, a gradual analysis will still report false positives anywhere its base static analysis is utilized and reports false positives. As a result, we plan to explore the aforementioned research questions, including the trade-off between gradual analyses reducing false positives and being conservative extensions of underlying static analyses.

References

- 1 Nathaniel Ayewah and William Pugh. The google findbugs fixit. In *Proceedings of the 19th international symposium on Software testing and analysis*, pages 241–252, 2010.

- 2 Johannes Bader, Jonathan Aldrich, and Éric Tanter. Gradual program verification. In *International Conference on Verification, Model Checking, and Abstract Interpretation*, pages 25–46. Springer, 2018.
- 3 Subarno Banerjee, Lazaro Clapp, and Manu Sridharan. Nullaway: Practical type-based null safety for java. *arXiv preprint arXiv:1907.02127*, 2019.
- 4 Mike Barnett, Manuel Fahndrich, Francesco Logozzo, and Diego Garbervet-sky. Annotations for (more) precise points-to analysis. In *IWACO 2007*, January 2007. URL: <https://www.microsoft.com/en-us/research/publication/annotations-for-more-precise-points-to-analysis/>.
- 5 Dan Brotherston, Werner Dietl, and Ondřej Lhoták. Granular: Gradual nullable types for java. In *Proceedings of the 26th International Conference on Compiler Construction*, CC 2017, pages 87–97, New York, NY, USA, 2017. ACM. doi:10.1145/3033019.3033032.
- 6 Patrice Chalin and Perry R James. Non-null references by default in java: Alleviating the nullity annotation burden. In *European Conference on Object-Oriented Programming*, pages 227–247. Springer, 2007.
- 7 Jong-Deok Choi, Keunwoo Lee, Alexey Loginov, Robert O’Callahan, Vivek Sarkar, and Manu Sridharan. Efficient and precise datarace detection for multithreaded object-oriented programs. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, PLDI ’02, page 258–269, New York, NY, USA, 2002. Association for Computing Machinery. doi:10.1145/512529.512560.
- 8 Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the 4th ACM Symposium on Principles of Programming Languages (POPL 77)*, pages 238–252, Los Angeles, CA, USA, January 1977. ACM.
- 9 Brian A Davey and Hilary A Priestley. *Introduction to lattices and order*. Cambridge university press, 2002.
- 10 Sam Estep, Jenna Wise, Jonathan Aldrich, Éric Tanter, Johannes Bader, and Joshua Sunshine. Gradual program analysis for null pointers, 2021. arXiv:2105.06081.
- 11 Facebook. Infer: A tool to detect bugs in java and c/c++/objective-c code before it ships. <https://fbinfer.com/>, 2019. Accessed: 2019-10-28.
- 12 Facebook. Eradicate. <https://fbinfer.com/docs/checker-eradicate>, 2020. Accessed: 2021-1-10.
- 13 Robert Bruce Findler and Matthias Felleisen. Contracts for higher-order functions. In *Proceedings of the 7th ACM SIGPLAN Conference on Functional Programming (ICFP 2002)*, pages 48–59, Pittsburgh, PA, USA, 2002. ACM.
- 14 Ronald Garcia, Alison M. Clark, and Éric Tanter. Abstracting gradual typing. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL ’16, pages 429–442, New York, NY, USA, 2016. ACM. doi:10.1145/2837614.2837670.
- 15 Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. Why don’t software developers use static analysis tools to find bugs? In *Proceedings of the 2013 International Conference on Software Engineering*, pages 672–681. IEEE Press, 2013.
- 16 Gary A Kildall. A unified approach to global program optimization. In *Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 194–206. ACM, 1973.
- 17 Kenneth Knowles and Cormac Flanagan. Hybrid type checking. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 32(2):1–34, 2010.
- 18 Bertrand Meyer. *Eiffel: The Language*. Prentice Hall, 1992.
- 19 Abel Nieto, Marianna Rapoport, Gregor Richards, and Ondřej Lhoták. Blame for null. In *European Conference on Object-Oriented Programming*, 2020.
- 20 Peter W O’Hearn. Incorrectness logic. *Proceedings of the ACM on Programming Languages*, 4(POPL):1–32, 2019.

- 21 Matthew M Papi, Mahmood Ali, Telmo Luis Correa Jr, Jeff H Perkins, and Michael D Ernst. Practical pluggable types for java. In *Proceedings of the 2008 international symposium on Software testing and analysis*, pages 201–212, 2008.
- 22 Jeremy G Siek and Walid Taha. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop*, volume 6, pages 81–92, 2006.
- 23 Jeremy G Siek, Michael M Vitousek, Matteo Cimini, and John Tang Boyland. Refined criteria for gradual typing. In *LIPICs-Leibniz International Proceedings in Informatics*, volume 32. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015.
- 24 Jenna Wise, Johannes Bader, Cameron Wong, Jonathan Aldrich, Éric Tanter, and Joshua Sunshine. Gradual verification of recursive heap data structures. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–28, 2020.
- 25 Dana N Xu. Hybrid contract checking via symbolic simplification. In *Proceedings of the ACM SIGPLAN 2012 workshop on Partial evaluation and program manipulation*, pages 107–116, 2012.