Gradual Program Analysis

SAM ESTEP, Liberty University, USA
JENNA WISE, Carnegie Mellon University, USA
JONATHAN ALDRICH, Carnegie Mellon University, USA
ÉRIC TANTER, University of Chile, Chile
JOHANNES BADER, Facebook, USA
JOSHUA SUNSHINE, Carnegie Mellon University, USA

The designers of static analyses used in industry often try to reduce the number of false positives reported by the analysis through increased engineering effort, user-provided annotations, and/or weaker soundness guarantees. To produce a static analysis with little engineering effort, reduced false positives, and strong soundness guarantees in a principled way, we adapt the "Abstracting Gradual Typing" framework to the abstract-interpretation based program analysis setting. As a case study, we take a simple static dataflow analysis that relies on user-provided annotations and has nullability lattice $N \sqsubset \top$ (where N means "definitely not null" and $\top$ means "possibly null") and extend it by adding ? as a third abstract value. The question mark explicitly represents "optimistic uncertainty" in the analysis itself, supporting a formal soundness property and the "gradual guarantees" laid out in the gradual typing literature. To evaluate our gradual null-pointer analysis, we implement it as a Facebook Infer checker and compare it against the existing null checkers in Facebook Infer. A preliminary set of experiments show evidence of reduced false positives. We then generalize this example into a system that gradualizes any dataflow analysis in the same way: augmenting the lattice while retaining the properties that permit it to be used in the standard dataflow analysis fixpoint algorithm.

CCS Concepts: • **Software and its engineering → General programming languages**; • **Social and professional topics** → *History of programming languages*;

Additional Key Words and Phrases: gradual typing, gradual verification, dataflow analysis, null pointer analysis

ACM Reference Format:
Sam Estep, Jenna Wise, Jonathan Aldrich, Éric Tanter, Johannes Bader, and Joshua Sunshine. 2020. Gradual Program Analysis. 1, 1 (January 2020), 13 pages.

1 INTRODUCTION

A significant proportion of software defects can be eliminated by a relatively specialized set of analyses. For instance, 53.8–57.1% of crash-causing defects in Mozilla and Apache Web Server as of 2006 were found to be memory bugs, and 37.2–41.7% of those were specifically NULL pointer dereferences [Li et al. 2006]. Static analysis could be used to find these defects, but sound tools—that is, those without false negatives—do not scale well, and burden developers with too many false positives [Emanuelsson and Nilsson 2008], which cause inconsistent use of static analysis tools in industry [Johnson et al. 2013]. To avoid these issues, state-of-the-art null-pointer analysis tools make use of modularity, user-provided annotations, and strategic unsoundness [Banerjee et al. 2019]. However, this unsoundness is generally unformalized, which makes it difficult to extend to analyses for other defects besides NULL pointer dereferences. In response, this work lays the groundwork for a general framework of gradual static analyses using concepts from the gradual typing literature.[1]

[1] An extended abstract for this work appeared in the SPLASH 2019 proceedings [Estep 2019].

2020. XXXX-XXXX/2020/1-ART
https://doi.org/

Recent work by Bader et al. [2018] draws on research in gradual typing [Garcia et al. 2016; Siek and Taha 2007, 2006] to produce a sound and more user-friendly approach to formal verification called *gradual verification*. Gradual verification extends a static verification system with support for *imprecise specifications* – a partial specification joined with ? representing unspecified information, similarly to gradual refinement types [Lehmann and Tanter 2017] – using Garcia et al. [2016]'s *Abstracting Gradual Typing* framework. The resulting verification system warns only about inconsistencies between specifications and code; it does not produce warnings due to missing information in specifications. Instead, missing information is dynamically verified. Gradual verification adheres to gradual guarantees (inspired by corresponding gradual typing properties formulated by Siek et al. [2015]) that ensure developers can choose their desired level of precision without artificial constraints imposed by the verification technology.

This paper explores a novel territory for gradual program reasoning: the gradual program analysis setting produces sound analyses that reduce false positives through user- or tool-provided annotations and runtime checks in a principled way. We present a case study that consists of gradualizing a simple null-pointer analysis based on *abstract interpretation* [Cousot and Cousot 1977]. Its lattice is extended to support two conceptions of uncertainty at once. The original lattice already permits *pessimistic* uncertainty, in which the analysis assumes the worst and issues a static warning whenever it determines that the program *may* go wrong. Our larger set also permits *optimistic* uncertainty, in which the analysis issues no static warning but instead inserts a runtime check to prevent the program from going wrong. The choice between pessimistic and optimistic uncertainty is determined by user-provided annotations.

Introducing optimistic uncertainty must necessarily reduce false positives, which is our primary goal as mentioned above. Our experiments using a custom Facebook Infer checker on Java code show that we reduce false positives not just in comparison to sound static analyses, but also in comparison to unsound analyses used in industry. The other benefit of our approach is that the programmer can choose where to use optimistic uncertainty and where to use pessimistic uncertainty by placing annotations in their code, so they can choose to get analysis results where it would be useful, and avoid most false positives everywhere else.

The rest of this paper is structured as follows. First, in section 2, describe our gradual program analysis framework outside the context of any specific analysis, along with some of the formal properties that it satisfies. Then in section 3, we take a simple, standard null-pointer analysis on a Java-like language, and gradualize it using our framework. Finally, in section 4, we present some preliminary empirical findings related to the gradual null-pointer analysis, before listing related work in section 5 and concluding with a discussion of future work in section 6.

2 GENERAL FRAMEWORK AND PROPERTIES

2.1 Static Analysis

We start with a set Var of possible variables. We then assume that we have a programming language, which consists of (among other things) a family Inst_{Ann} of instruction sets, parametrized by a nonempty set Ann of annotations. Annotations do not affect the behavior of the program, but will be used by the static (and later, gradual) analysis. A program in this language is a directed graph on the set of vertices Vert, with a mapping $\text{inst} : \text{Vert} \to \text{Inst}_{\text{Ann}}$. This graph is the control-flow graph of the program, so essentially, an edge from $u \in \text{Vert}$ to $v \in \text{Vert}$ means that, if the program is currently at u, execution could proceed to vertex v after u is executed. If a vertex has multiple outgoing edges, it is a branch point; the edge that is actually taken when the program executes would depend on the program state and the concrete semantics of the language.

We also assume that we have a static analysis for this language, which has three parts:

```
for Instruction i in program
    input[i] = output[i] = ⊤
output[programStart] = initialDataflowInformation
worklist = { firstInstruction }

while worklist is not empty
    take an instruction i off the worklist
    input[i] = ⊔_{k∈preds(i)} output[k]
    newOutput = F(i, input[i])
    if newOutput ≠ output[i]
        output[i] = newOutput
        for Instruction j in succs(i)
            add j to worklist
```

Fig. 1. Kildall's worklist algorithm for the null-pointer analysis.

- A semilattice $\textsc{Abst}$ of abstract values with partial order $\sqsubseteq$ and join function $\sqcup$
- A family of flow functions $\textsc{flow}_{\textsc{Ann}} : \textsc{Inst}_{\textsc{Ann}} \times \textsc{State}_{\textsc{Ann}} \to \textsc{State}_{\textsc{Ann}}$ (where $\textsc{State}_{\textsc{Ann}} = \textsc{Var} \rightharpoonup \textsc{Ann}$), parametrized by the set of annotations $\textsc{Ann}$, where it can be assumed that $\textsc{Ann} \supseteq \textsc{Abst}$
- A family of safety functions $\textsc{safe}_{\textsc{Ann}} : \textsc{Var} \times \textsc{Inst}_{\textsc{Ann}} \to \textsc{Abst}$, where again it can be assumed that $\textsc{Ann} \supseteq \textsc{Abst}$

From these three analysis parts, we take an individual program with $\textsc{Ann} = \textsc{Abst}$ and use Kildall's worklist algorithm [Kildall 1973], outlined in Figure 1, to arrive at our analysis fixpoint $\textsc{fixp} : \textsc{Vert} \to \textsc{State}_{\textsc{Ann}}$. At a high level, we start by mapping each vertex $\textsc{Vert}$ to the empty map $\varnothing \in \textsc{State}_{\textsc{Ann}}$; then repeatedly

(1) choose some vertex $v \in \textsc{Vert}$,
(2) use our current mapping $\pi : \textsc{Vert} \to \textsc{State}_{\textsc{Ann}}$ to get $\sigma = \pi(v) \in \textsc{State}_{\textsc{Ann}}$,
(3) run that through our flow function to get $\sigma' = \textsc{flow}_{\textsc{Ann}}(\textsc{inst}(v), \sigma)$,
(4) update to get π' where for each edge from v to some node u we have $\pi'(u) = \pi(u) \sqcup \sigma'$.

In that last step we elementwise let $(\sigma_1 \sqcup \sigma_2)(x) = \sigma_1(x) \sqcup \sigma_2(x)$ for $\sigma_1, \sigma_2 \in \textsc{State}_{\textsc{Ann}}$.

After this algorithm terminates, the final results $\textsc{fixp}$ tell us, for each program point in $v \in \textsc{Vert}$ and each variable in $x \in \textsc{Var}$, some abstract value $a \in \textsc{Abst}$ that "contains" all the possible values which x could take on at v when the program is run. If $a \not\sqsubseteq \textsc{safe}_{\textsc{Ann}}(x, \textsc{inst}(v))$, then the analysis has deemed that variable "unsafe" at that point, and we issue a warning to the programmer. The analysis should also come with a proof that some semantic property holds if no warnings are issued.

Our framework lifts $\textsc{Abst}$ to a larger set $\widetilde{\textsc{Abst}}$ such that $\textsc{Abst}$ is a subsemilattice of $\widetilde{\textsc{Abst}}$, then uses it to define a new static analysis using this larger set of abstract values. This lifting is detailed in the following subsection. Also as explained in Subsection 2.2, we obtain a lifted ordering $\widetilde{\sqsubseteq}$ and join $\widetilde{\sqcup}$ on $\widetilde{\textsc{Abst}}$, which we can then use to lift the rest of the analysis.

We now accept programs using the extended set of annotations $\textsc{Ann} = \widetilde{\textsc{Abst}}$, and again apply Kildall's fixpoint algorithm to arrive at $\widetilde{\textsc{fixp}} : \textsc{Vert} \to \textsc{State}_{\widetilde{\textsc{Abst}}}$. As we will find, the pair $(\widetilde{\textsc{Abst}}, \widetilde{\sqcup})$ is a semilattice in the same way that $(\textsc{Abst}, \sqcup)$ is, so we can simply replace $\sqcup$ with $\widetilde{\sqcup}$ in step 4 of the inner loop. Then after the fixpoint algorithm terminates, we use $\widetilde{\sqsubseteq}$ instead of $\sqsubseteq$ to determine when to issue a static warning.

2.2 Semilattice Lifting

The AGT methodology provides the general setup for lifting our semilattice. To give meaning to gradual abstract elements from $\widetilde{\textsc{Abst}} \supseteq \textsc{Abst}$, AGT requires an injective function $\gamma : \widetilde{\textsc{Abst}} \to \mathcal{P}^+(\textsc{Abst})$. To ensure that the gradual analysis is a conservative extension of the static one, it must be the case that $\gamma(a) = \{a\}$ whenever $a \in \textsc{Abst}$. To maximize flexibility of the gradual analysis, we must have some element $? \in \widetilde{\textsc{Abst}}$ such that $\gamma(?) = \textsc{Abst}$. This γ then automatically induces lifted predicates such as $\widetilde{\sqsubseteq}$:

$$\widetilde{a} \mathrel{\widetilde{\sqsubseteq}} \widetilde{b} \quad \text{iff} \quad \exists a \in \gamma(\widetilde{a}), b \in \gamma(\widetilde{b}) \quad \text{s.t.} \quad a \sqsubseteq b$$

Then as mentioned above, this lifted ordering appears in the *second* stage of the analysis: given a table of abstract values for each variable at each program point, how do we determine when to give a static warning? But for the *first* stage, $\widetilde{\sqsubseteq}$ is insufficient. Observe that $? \mathrel{\widetilde{\sqsubseteq}} a \mathrel{\widetilde{\sqsubseteq}} ?$ for all $a \in \widetilde{\textsc{Abst}}$, so $\widetilde{\sqsubseteq}$ is not a partial order. Thus we cannot use $\widetilde{\sqsubseteq}$ to induce a join for use in the fixpoint algorithm.

To address this, we must be able to lift functions on $\widetilde{\textsc{Abst}}$, not just predicates. We are given a function $\sqcup : \textsc{Abst} \times \textsc{Abst} \to \textsc{Abst}$ and two elements $\widetilde{a}, \widetilde{b} \in \widetilde{\textsc{Abst}}$, and we want to produce some $\widetilde{a} \mathbin{\widetilde{\sqcup}} \widetilde{b} \in \widetilde{\textsc{Abst}}$. Via AGT, our first step is to construct the set $\{a \sqcup b : a \in \gamma(\widetilde{a}) \wedge b \in \gamma(\widetilde{b})\} \in \mathcal{P}^+(\textsc{Abst})$. Thus we need a function $\alpha : \mathcal{P}^+(\textsc{Abst}) \to \widetilde{\textsc{Abst}}$. In AGT, this α forms a Galois connnection with γ, so the following must hold:

$$\begin{aligned}
&\text{For any } \widehat{a} \in \mathcal{P}^+(\textsc{Abst}) \text{ and } \widetilde{b} \in \widetilde{\textsc{Abst}},\\
&\quad (1)\ \widehat{a} \subseteq \gamma(\alpha(\widehat{a})) \quad \text{and}\\
&\quad (2)\ \widehat{a} \subseteq \gamma(\widetilde{b}) \implies \gamma(\alpha(\widehat{a})) \subseteq \gamma(\widetilde{b}).
\end{aligned}$$

It can be shown that if α exists, it must be unique. Specifically,

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

where γ^{-1} is well-defined because γ is injective, but is only a partial function because γ is not necessarily surjective; thus, α might not necessarily exist. But if it does, we can then define the lifted join

$$\widetilde{a} \mathbin{\widetilde{\sqcup}} \widetilde{b} = \alpha(\{a \sqcup b : a \in \gamma(\widetilde{a}) \wedge b \in \gamma(\widetilde{b})\})$$

to use in the fixpoint algorithm.

So far, we have glossed over the construction of $\widetilde{\textsc{Abst}}$ itself, other than the fact that it must be a superset of $\textsc{Abst} \cup \{?\}$. It should now be clear that the details of this construction are an important design choice, since a poor choice of $\widetilde{\textsc{Abst}}$ can prevent the existence of α. Notice also that the characteristics of α determine the characteristics of $\widetilde{\sqcup}$, which are used in the proofs of termination and soundness of the analysis:

- *Commutativity:* $a \sqcup b = b \sqcup a$, so for instance, the analysis is the same regardless of whether the "then" branch or the "else" branch of an if-statement is analyzed first.
- *Idempotency:* $a \sqcup a = a$, so for instance, the analysis remains the same when an if-then-else-statement with both empty branches is inserted.
- *Transitivity:* $a \sqcup (b \sqcup c) = (a \sqcup b) \sqcup c$, so for instance, the analysis is the same regardless of whether a nested if-clause conditions on x first and then y, or the other way around.

In fact, these are exactly the conditions for $\sqcup$ to be the join operation of a semilattice structure. Furthermore, for the fixpoint computation to terminate, that semilattice must have finite height.

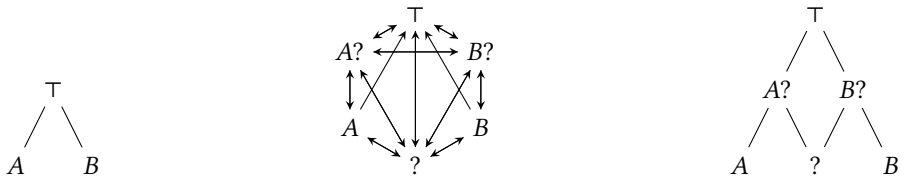

Fig. 2. *Left:* The original (unlifted) semilattice. *Middle:* The lifted semilattice ordering, where each directed edge $a \longrightarrow b$ means $a \mathrel{\widetilde{\sqsubseteq}} b$. (Self-loops are omitted.) This lifted relation is no longer a partial order. *Right:* The semilattice structure induced by the lifted join $\widetilde{\sqcup}$. Specifically, this is the Hasse diagram of the partial order $\{(a, b) : a \mathbin{\widetilde{\sqcup}} b = b\}$.

We have already mentioned that the lifted relation $\widetilde{\sqsubseteq}$ does *not* form a semilattice. However, the surprising, and novel, result of this work is that the lifted join $\widetilde{\sqcup}$ *does* form a semilattice, but only if the lifted set $\widetilde{\textsc{Abst}}$ is chosen correctly.

Commutativity is trivially satisfied. Our set $\widetilde{\textsc{Abst}}$ cannot contain any element $\widetilde{a}$ such that $\gamma(\widetilde{a})$ is not a subsemilattice of Abst, since that breaks idempotency. We also cannot in general choose $\widetilde{\textsc{Abst}} = \textsc{Abst} \cup \{?\}$, because this produces a non-associative join on even simple semilattices such as $\textsc{Abst} = \{A, B, \top\}$ where $A, B \sqsubset \top$ but $A \not\sqsubseteq B$ and $B \not\sqsubseteq A$. (See Figure 2 for an illustration of a "correct" lifting of this semilattice, which adjoins more than just one element.) On the other hand, we cannot let $\widetilde{\textsc{Abst}}$ contain an element for every subsemilattice (or even just for every convex subsemilattice; see Cheong and Jones [2003]) of $\widetilde{\textsc{Abst}}$, since then $\widetilde{\textsc{Abst}}$ with $\widetilde{\sqcup}$ might not have finite height even if Abst with $\sqcup$ has finite height.

To satisfy all of our constraints, we choose

$$\widetilde{\textsc{Abst}} = \textsc{Abst} \cup \{?\} \cup \{a? : a \in \textsc{Abst}\} \quad \text{where} \quad \gamma(a?) = \{b \in \textsc{Abst} : a \sqsubseteq b\}$$

since it can be shown that if Abst has finite height h, then this choice of $\widetilde{\textsc{Abst}}$ always has an abstraction function, yielding a $\widetilde{\sqcup}$ operation which induces a semilattice structure with height $h + 1$. Note that in this lifting,

$$\top? = \top \quad \text{because} \quad \gamma(\top?) = \{a \in \textsc{Abst} : \top \sqsubseteq a\} = \{\top\} = \gamma(\top),$$

and similarly if L has a bottom element $\bot$ then

$$\bot? = ? \quad \text{because} \quad \gamma(\bot?) = \{a \in \textsc{Abst} : \bot \sqsubseteq a\} = \textsc{Abst} = \gamma(?).$$

This means that if Abst has n elements then $\widetilde{\textsc{Abst}}$ has either $2n$ or $2n - 1$ elements.

2.3 Runtime Checks

The above theory defines the static part of our gradual analysis framework. It is usable on its own as a method to reduce false positives by providing the programmer with a broader set of annotations which support both pessimistic and optimistic uncertainty. Our prototype, described in Section 4, only implements the static part of the analysis, and does not insert runtime checks because program transformation is not supported by the Infer framework. However, it is "unsound", in that the absence of warnings from a gradual analysis does not provide the same semantic guarantee provide by the underlying static analysis. To remedy this, one can add runtime checks based on the analysis results. This subsection will not be quite as rigorous as 2.1, but should be straightforward enough given the rest of the theory already described.

First, we concretize our set of absolute values via a set of concrete (runtime) values Val and a concretization function $\textsc{conc} : \textsc{Abst} \to \mathcal{P}^+(\textsc{Val})$. We will then assume that, for each $a \in \textsc{Abst}$, the

predicate $\text{IN} \subseteq \text{VAL} \times \text{ABST}$ defined by

$$c \text{ IN } a \quad \Longleftrightarrow \quad c \in \text{CONC}(a)$$

is computable. We also assume that we have $\text{CHECK}_{\text{ANN}} : \text{VAR} \times \text{ABST} \to \text{INST}_{\text{ANN}}$ such that the semantics of $\text{CHECK}_{\text{ANN}}(x, a)$ are to get the value $c \in \text{VAL}$ currently held locally by x, then if c IN a, continue along the control flow graph; or if $\neg(c \text{ IN } a)$, step into a dedicated error state.

After we run our fixpoint algorithm to get our analysis results $\widetilde{\text{FIXP}}$ and ensuring that there are no static warnings, we transform the input program by inserting these runtime checks. Specifically, for every $v \in \text{VERT}$, let $\sigma = \widetilde{\text{FIXP}}(v)$; then for every $x \in \text{VAR}$, let $\widetilde{a} = \sigma(x)$. Let $b = \bigsqcup \gamma(\text{SAFE}_{\widetilde{\text{ABST}}}(x, \text{INST}(v)))$. If it isn't the case that $a \sqsubseteq b$ for every $a \in \gamma(\widetilde{a})$, then insert a new vertex u before v such that $\text{INST}(u) = \text{CHECK}_{\text{ANN}}(x, b)$. Iterate this process until no troublesome points remain.

If (as in our null-pointer case study) the set of concrete values is the set of pointers, and ABST only distinguishes null from non-null, this is efficiently computable. But for instance, an analysis of a language with higher-order functions would be more difficult to gradualize, since IN would be undecidable. This problem is not intractable, and has been discussed at length in the gradual typing literature. However, it is by no means trivial, and would require further research to adapt it to our dataflow analysis setting.

3 CASE STUDY: A SIMPLE NULL-POINTER ANALYSIS

We will now illustrate our general framework in the context of null-pointer analysis. First, recall that Section 2 defines a program as its control flow graph. A complete illustration of our formal framework would fully define the instruction set for this control-flow-graph-based IR and and a conversion from source code to this IR. However, for brevity, we will instead just display the Java source code (see Figure 3), and include portions of the explicit formalism where it would be relevant.

An example execution of this code might be as follows:

```
java Bucket.java maple pecan grape almond walnut cherry apricot
```

The output would look something like this:

```
2 of apricot wood
5 of cherry pie
1 of walnut wood
5 of almond pie
5 of grape pie
5 of pecan pie
7 of maple wood
0 of apple wood
Exception in thread "main" java.lang.NullPointerException
        at Bucket.main(Bucket.java:45)
```

This program handles `Buckets`, each of which can hold some `amount` of either chopped wood of some `woodType`, or baked pies of some `pieFlavor`. Thus, its contents can be either `edible` or not. A bucket can also be nested inside of some `parent` bucket. The `chop` method adds to the bucket one unit of a specific `woodType`, discarding the previous contents of the bucket if they are not of the same `woodType`. The `bake` method adds five pies of a specific `pieFlavor` to the bucket, discarding any previous contents because even pies of the same flavor would be less fresh than these new pies. The `main` method takes in a list of plant types, and for each one, repeatedly either chops wood from that plant of that type or bakes pies using the fruit from that plant. It also nests all of these

```
1  class Bucket {
2    @Nullable Bucket parent = null;
3    @NonNull Boolean edible = Boolean.FALSE;
4    String woodType = "apple";
5    String pieFlavor = null;
6    @NonNull Integer amount = 0;
7
8    void chop(@NonNull String woodType) {
9      this.edible = Boolean.FALSE;
10     this.pieFlavor = null;
11     if (!woodType.equals(this.woodType))
12       this.amount = 0;
13     this.woodType = woodType;
14     this.amount++;
15   }
16
17   void bake(@NonNull String pieFlavor) {
18     this.edible = Boolean.TRUE;
19     this.woodType = null;
20     this.pieFlavor = pieFlavor;
21     this.amount = 5;
22   }
23
24   public static void main(@NonNull String[] args) {
25     java.util.List<Bucket> buckets = new java.util.LinkedList<>();
26     buckets.add(new Bucket());
27     for (String name : args) {
28       Bucket bucket = new Bucket();
29       double cap = 10*Math.random();
30       for (int i = 0; i < cap; i++) {
31         if (Math.random() < 0.5)
32           bucket.chop(name);
33         else
34           bucket.bake(name);
35       }
36       bucket.parent = buckets.get(0);
37       buckets.add(0, bucket);
38     }
39     System.out.print(buckets.get(0).amount);
40     for (Bucket bucket : buckets) {
41       String type = bucket.edible
42         ? bucket.pieFlavor + "␣pie"
43         : bucket.woodType + "␣wood";
44       System.out.println("␣of␣" + type);
45       System.out.print(bucket.parent.amount);
46     }
47   }
48 }
```

Fig. 3. An example program.

$$
\begin{array}{rcl}
x, y, z & \in & \textsc{Var} \\
a, b & \in & \textsc{Ann} \\
m & \in & \textsc{Method} \\
f & \in & \textsc{Field} \\
\kappa & \in & \textsc{Class} \\
n & \in & \mathbb{N} \\
\textsc{Inst}_{\textsc{Ann}} & ::= & \texttt{method}\ m(x_1@a_1, \ldots, x_n@a_n) \mid x := \texttt{null} \mid x := \texttt{true} \\
& & \mid x := \texttt{false} \mid x := n \mid x := \kappa.f@a \mid x := y.f@a \\
& & \mid x.f@a := y \mid x := \texttt{new}\ \kappa \\
& & \mid x := \kappa.m@a(y_1@b_1, \ldots, y_n@b_n) \\
& & \mid x := y.m@a(z_1@b_1, \ldots, z_n@b_n) \mid \texttt{if}\ x \mid \texttt{if not}\ x \\
& & \mid \texttt{return}\ x@a \mid \ldots
\end{array}
$$

Fig. 4. Some portions of $\textsc{Inst}_{\textsc{Ann}}$ for our null-pointer analysis case study.

buckets inside each other. Finally, in the second loop, it goes through and displays the contents of each bucket, using the nestedness to access the amount of items in the next bucket before reaching it in the iteration. The reader will note that this results in a `NullPointerException` at the end of the list.

To analyze this program for null pointers, we will start with a pure static analysis, and then gradualize it via the framework from subsection 2.1. The set Var is what one would expect; after that, the first step is to define our set of instructions, of which we show part in Figure 4. (The sets of Method, Field, and Class names are straight forward just as Var was.) For an example of a control-flow graph in this language, see the translation of the `chop` method in Figure 5. Each method forms its own separate connected component of the control-flow graph.

Then for the analysis itself, we must first have a semilattice of abstract values (see Figure 6). In this case, we need only two: $\top$, which can represent any pointer, and N, which can represent any non-`null` pointer. In our syntax, we represent these as `@Nullable` and `@NonNull` respectively. After we gradualize this semilattice, we will have a third value ?, which we represent via the annotation `@Unknown` (omitted by convention in the source code, but present in the control-flow graph IR, as seen in Figure 5). Then to complete the initial static analysis, we also need a flow function and a safety function. Partial versions of $\textsc{flow}_{\textsc{Ann}}$ and $\textsc{safe}_{\textsc{Ann}}$ are shown in Figure 7 and Figure 8.

The last step in this case study is to lift Abst to $\widetilde{\textsc{Abst}}$ and apply the resulting gradual analysis. The lifted semilattice itself is shown in Figure 6. Here are some observations of the result of analyzing the program from Figure 3 via $\widetilde{\textsc{Abst}}$, flow, and safe:

- Since the fields `edible` and `amount` are `@NonNull`, the analysis doesn't complain when they are (implicitly) deferenced in lines 14 and 41.
- Since the parameter `woodType` in the `chop` method is `@NonNull`, the analysis doesn't complain when it is dereferenced (to call `equals`) in line 11.
- Since the fields `woodType` and `pieFlavor` are not explicitly annotated, they are implicitly marked as `@Unknown` (that is, "?") by the analysis. Intuitively, this means that the analysis knows that they could possibly be null, so it doesn't complain when they are explicitly assigned `null` values in lines 10 and 19; but on the other hand, since they aren't explicitly marked as `@Nullable`, the analysis also doesn't complain when they are dereferenced in lines 42 and 43 for concatenation.

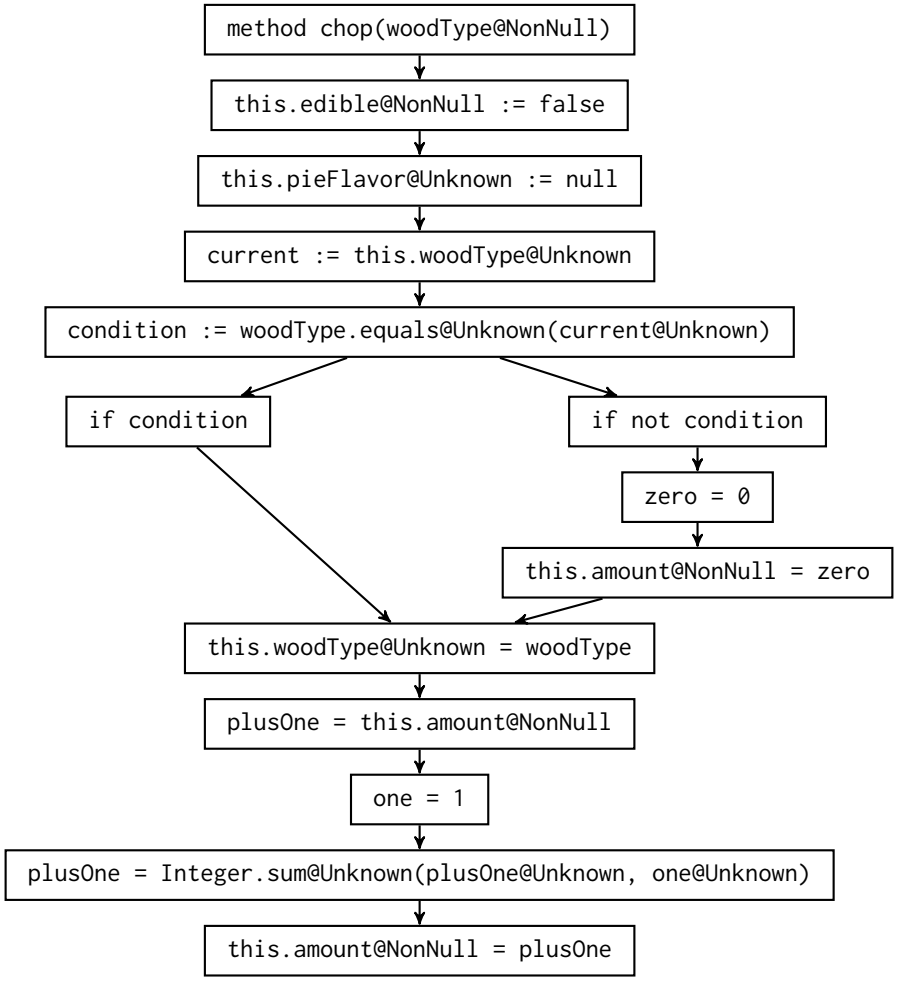

Fig. 5. A possible control-flow graph for the chop method.

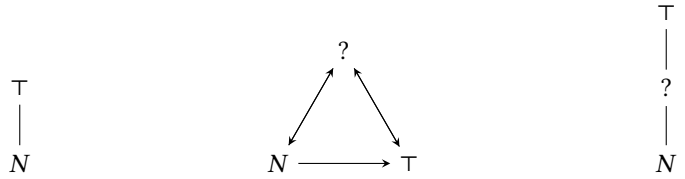

Fig. 6. *Left:* The original null-pointer semilattice. *Middle:* The lifted semilattice ordering, where each directed edge $a \longrightarrow b$ means $a \mathrel{\widetilde{\sqsubseteq}} b$. (Self-loops are omitted.) *Right:* The semilattice structure induced by the lifted join $\widetilde{\sqcup}$.

- The `for` loops on lines 27–38 and lines 40–46 implicitly use iterators, which are part of the Java standard library and thus not annotated with `@Nullable` or `@NonNull`. Thus, all their method parameters and return values are given the "?" (that is, `@Unknown`) annotations by

$$\begin{aligned}
\textsc{flow}_{\textsc{Ann}}(x := \texttt{null}, \sigma) &= \sigma[x \mapsto \top] \\
\textsc{flow}_{\textsc{Ann}}(x := y.f@a, \sigma) &= \sigma[x \mapsto a; y \mapsto N] \\
\textsc{flow}_{\textsc{Ann}}(x.f@a := y, \sigma) &= \sigma[x \mapsto N] \\
\textsc{flow}_{\textsc{Ann}}(x := \texttt{new}\ \kappa, \sigma) &= \sigma[x \mapsto N] \\
\textsc{flow}_{\textsc{Ann}}(x := y.m@a(z_1@b_1, \ldots, z_n@b_n), \sigma) &= \sigma[x \mapsto a]
\end{aligned}$$

Fig. 7. Part of a possible flow function for a null-pointer analysis.

$$\begin{aligned}
\textsc{safe}_{\textsc{Ann}}(y, x := y.f@a) &= N \\
\textsc{safe}_{\textsc{Ann}}(x, x.f@a := y) &= N \\
\textsc{safe}_{\textsc{Ann}}(y, x.f@a := y) &= a \\
\textsc{safe}_{\textsc{Ann}}(z_i, x := y.m@a(z_1@b_1, \ldots, z_n@b_n)) &= b_i \\
\textsc{safe}_{\textsc{Ann}}(x, \texttt{return}\ x@a) &= a
\end{aligned}$$

Fig. 8. Part of a possible safety function for a null-pointer analysis.

default, so the analysis doesn't complain when `name` is passed as a `NonNull` parameter in lines 32 and 34, or when `bucket` is dereferenced on lines 41–43 and 45.

- Finally, since the field `parent` is explicitly `@Nullable`, the analysis complains when it it dereferenced on line 45.

As noted in the last bullet point, the analysis thus only raises one warning for this code, on line 45, and that warning turns out to be a true positive, as evidenced by the `NullPointerException` that results when this program is run.

4 PRELIMINARY EMPIRICAL EVALUATION

We used the abstract interpretation framework in Facebook's Infer tool to build a prototype of a gradual null pointer analyzer for Java, which we'll call "Graduator", based on the development presented above. To evaluate this prototype, we used the 18 repositories which Uber used to evaluate their NullAway analysis tool [Banerjee et al. 2019]. Facebook has two existing null-pointer checkers in Infer, called Eradicate and Nullsafe; we ran those along with Graduator on the 15 repositories of those 18 which we were able to successfully build, with the following results:

- Eradicate gave 1489 static warnings.
- Nullsafe gave 654 static warnings.
- Graduator gave 228 static warnings.

These are all repositories for which NullAway reports no static errors, and Uber has found no instances of null-pointer dereferences caused by a false negative in their tool. According to Banerjee et al. [2019]: "NullAway aims to have *no false negatives in practice for code that it checks*, while reducing the annotation burden wherever possible. NullAway's checks to ensure `@NonNull` fields are properly initialized … are unsound, but also require far fewer annotations than a previous sound checker …. Similarly, NullAway unsoundly assumes that methods are pure, i.e., side-effect-free and deterministic …. In both cases, we hvae validated that neither source of unsoundness seems to

lead to real-world NPEs for Uber's Android apps, based on crash data from the field." Thus, it seems fairly reasonable to assume that these are all false positives. We examined all these 2371 warnings by hand, and found only 57 that could possibly be true positives; all the rest were due to systematic imprecision in the analysis tools.

In these results we see weak evidence that our Graduator tool tends to produce fewer false positives than Facebook's existing null-pointer analysis tools. However, since the analyzed repositories were already believed not to contain null pointer errors, it is still very unclear whether Graduator is preferable to Eradicate or Nullsafe, since it is possible that Graduator has a significantly higher false negative rate, at least in the static portion. Experiments on buggy repositories should be performed to compare the false negative rates of Graduator with other tools.

5 RELATED WORK

We have already related our work to the most-closely related research, including work on abstract interpretation [Cousot and Cousot 1977], the fixpoint algorithm [Kildall 1973], gradual typing properties [Siek et al. 2015], constructing gradual type systems from static ones [Garcia et al. 2016], gradual verification [Bader et al. 2018], and current null-pointer analysis tools [Banerjee et al. 2019].

We empirically compared Graduator to existing Facebook Infer checkers in Section 4. Here we discuss the approaches behind similar tools. The Granullar type system from Brotherston et al. [2017] extends a static pluggable type system that enforces null-safety via `@Nullable` and `@NonNull` type annotations with support for the `@Dynamic` type similar to ?. The resulting type system generates runtime checks whenever a `@Dynamic` reference is pseudo-assigned to a @NonNull reference. This guarantees that null-pointer exceptions cannot occur in checked code when interacting with unchecked software components. Our approach differs from Granullar's approach in that we do not draw boundaries between checked and unchecked code; every field, argument, and return value is either implicitly ?, explicitly N or $\top$. Thus, our approach is more fine-grained.

NullAway [Banerjee et al. 2019] reduces the annotation burden of static pluggable type checking for null-pointer exceptions through targeted unsound assumptions. They aim for no false negatives in practice on checked code. Of course, this differs from our approach by not trying to be sound. Instead of letting optimistic assumptions go unchecked, we inject corresponding runtime checks to verify them.

Some related work in gradual typing is as follows. The concept of gradual union types has been explored by Toro and Tanter [2017]. This is related in that null-pointer analyses can be thought of in terms of flow-sensitive union types. Toro and Tanter [2017] provides a novel construct yielding the benefits from both tagged and untagged unions. Interestingly, when refined using the specific example of nullable pointers, these new gradual unions are essentially the same as the treatment of nullable pointers in modern programming languages: static checking for nullability is nearly absent, so checks must be performed at runtime. In comparison, our approach provides a bit more static information.

Another related effort in gradual typing, is recent work on gradual refinement types [Lehmann and Tanter 2017]. In that approach, the AGT methodology is applied to a functional language in which types can be refined by logical predicates drawn from a decidable logic. The present work is in a different context, namely first-order imperative programs as opposed to higher-order pure functional programs. This difference has an impact on the technical development. Additionally, we provide a runtime semantics designed for the gradual program analysis setting (similar to work on gradual verification [Bader et al. 2018]), rather than adapting the evidence-tracking approach set forth by the AGT methodology and used for gradual refinement types.

6 CONCLUSION

While the AGT framework has been very successful in a variety of formal type systems, this is its first application to program analysis. We show how to gradualize an arbitrary dataflow analysis to demonstrate the applicability of the AGT framework in the analysis setting; this lays the groundwork for the gradualization of far more sophisticated analyses than the simple case study demonstrated above.

The draw of AGT in general is that it tends to yield results that are about as good as (or better than) what one would reach from an initial, intentional effort to gradualize the system. But AGT is essentially a mechanical process, so it removes a lot of the error-prone nature of the gradualization work. The fact that the prototype of our case study ends up behaving similarly to state-of-the-art tools like Facebook Infer is an encouraging indicator that this line of work may be fruitful, although more rigorous evaluation needs to be done in the near future.

As mentioned in section 4, more experiments should be performed to determine whether the static false negative rate of our approach is tolerable. However, these false negatives only make our tool unsound in the absence of dynamic checks. We have outlined a basic theory for insert runtime checks, but it would be desirable to have a more general theory that doesn't require a computable "IN" predicate; perhaps such a theory should take inspiration from transient gradual typing [Vitousek et al. 2017]. Finally, while our gradual analysis framework is general and can be applied to arbitrary dataflow analyses, that generality could be made more composable by recasting our work as an abstract interpretation component in the sense developed by Keidel and Erdweg [2019].

ACKNOWLEDGMENTS

This material is based upon work supported by a Facebook Testing and Verification research award and the National Science Foundation under Grant No. CCF-1901033 and Grant No. DGE1745016. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

Johannes Bader, Jonathan Aldrich, and Éric Tanter. 2018. Gradual Program Verification. In *International Conference on Verification, Model Checking, and Abstract Interpretation*. Springer, 25–46.

Subarno Banerjee, Lazaro Clapp, and Manu Sridharan. 2019. NullAway: Practical Type-Based Null Safety for Java. *arXiv preprint arXiv:1907.02127* (2019).

Dan Brotherston, Werner Dietl, and Ondřej Lhoták. 2017. Granullar: Gradual Nullable Types for Java. In *Proceedings of the 26th International Conference on Compiler Construction (CC 2017)*. ACM, New York, NY, USA, 87–97. https://doi.org/10.1145/3033019.3033032

Kyeong Hee Cheong and Peter R Jones. 2003. The lattice of convex subsemilattices of a semilattice. In *Semigroup Forum*, Vol. 67. Springer, 111–124.

Patrick Cousot and Radhia Cousot. 1977. 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)*. Los Angeles, CA, USA, 238–252.

Pär Emanuelsson and Ulf Nilsson. 2008. A comparative study of industrial static analysis tools. *Electronic notes in theoretical computer science* 217 (2008), 5–21.

Samuel Estep. 2019. Gradual Program Analysis. In *Proceedings Companion of the 2019 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity (SPLASH Companion 2019)*. ACM, New York, NY, USA, 52–53. https://doi.org/10.1145/3359061.3361082

Ronald Garcia, Alison M. Clark, and Éric Tanter. 2016. Abstracting Gradual Typing. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '16)*. ACM, New York, NY, USA, 429–442. https://doi.org/10.1145/2837614.2837670

Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. 2013. Why don't software developers use static analysis tools to find bugs?. In *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press,

672–681.

Sven Keidel and Sebastian Erdweg. 2019. Sound and reusable components for abstract interpretation. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 176.

Gary A Kildall. 1973. A unified approach to global program optimization. In *Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. ACM, 194–206.

Nico Lehmann and Éric Tanter. 2017. Gradual Refinement Types. In *Proceedings of the 44th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2017)*. Paris, France, 775–788.

Zhenmin Li, Lin Tan, Xuanhui Wang, Shan Lu, Yuanyuan Zhou, and Chengxiang Zhai. 2006. Have things changed now?: an empirical study of bug characteristics in modern open source software. In *Proceedings of the 1st workshop on Architectural and system support for improving software dependability*. ACM, 25–33.

Jeremy Siek and Walid Taha. 2007. Gradual typing for objects. In *European Conference on Object-Oriented Programming*. Springer, 2–27.

Jeremy G Siek and Walid Taha. 2006. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop*, Vol. 6. 81–92.

Jeremy G Siek, Michael M Vitousek, Matteo Cimini, and John Tang Boyland. 2015. Refined criteria for gradual typing. In *LIPIcs-Leibniz International Proceedings in Informatics*, Vol. 32. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

Matías Toro and Éric Tanter. 2017. A Gradual Interpretation of Union Types. In *Static Analysis*, Francesco Ranzato (Ed.). Springer International Publishing, Cham, 382–404.

Michael M Vitousek, Cameron Swords, and Jeremy G Siek. 2017. Big types in little runtime: open-world soundness and collaborative blame for gradual type systems. In *ACM SIGPLAN Notices*, Vol. 52. ACM, 762–774.