SPLASH: U: Gradual Program Analysis

Sam Estep
College of Arts & Sciences
Liberty University
Lynchburg, VA, USA
sestep@liberty.edu

ABSTRACT

Annotations are a common and useful way to make static analysis tools more usable, but there is no standard way to analyze programs that have only partial annotations. We give a formal framework that transforms arbitrary static analyses assuming complete annotations into gradual analyses accepting partial annotations. Our approach is based on gradual typing, and thus combines static and dynamic analysis to give soundness guarantees via runtime checks. We satisfy a preestablished property called the gradual guarantee, ensuring that our system reacts predictably as the programmer adds annotations one-by-one. We implement a specific instance of our general framework as a static null-pointer checker in Facebook Infer, and find in a preliminary empirical investigation that our framework gives fewer statically-reported false positives than Infer's existing checkers.

KEYWORDS

gradual typing, gradual verification, dataflow analysis

1 INTRODUCTION

Static analysis tools have been successfully used on a large scale to find and fix software defects [1], but in general, they are "underused" in practice due to the number of false positives that they tend to produce [9]. Techniques to reduce these false positives include increased precision through ingenious engineering effort [12], increased modularity through programmer-provided annotations [4], and strategic unsoundness [3]. For instance, the documented Eradicate checker and newer `--nullsafe` checker in Facebook Infer make use of `@Nullable` and `@NonNull` annotations in Java code to analyze for null-pointer bugs. These annotations in particular are very common in real-world code because so many different analysis tools make use of them [3], but it is rare for a codebase to be perfectly and completely annotated, so the analysis tool must decide whether to interpreted the lack of annotation as an implicit `@Nullable` or `@NonNull` annotation, or as something else entirely. Review of the source code of Infer's `--nullsafe` checker reveals the latter approach, but without a consistent set of principles for treating the difference between an annotation and the lack thereof. This paper describes the gradual program analysis project [6, 7], which provides a sound, principled approach to analysis of partially-annotated codebases using techniques from gradual typing [2, 8, 11].

> *Example.* The C program shown in Figure 1 illustrates the motivation for a sound analysis system where "no annotation" has a distinct meaning. When we use a tool to analyze this code, we might want some sort of static warning if, for instance, line 6 were missing, so we might put a `@Nullable` annotation on the `str` parameter. But we might not want a static warning on usages like line 15, so we might not annotate the return value of `reverse` with `@Nullable`, because then to appease the static analysis we would need to check its return value for `NULL` every time we use it. Yet even if we don't want to get a static warning on line 15, we also don't really want line 16 to segfault, so it is insufficient to simply ignore 16 entirely.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4
5  char* reverse(char* str) {
6    if (!str) return NULL;
7    int len = strlen(str);
8    char* rev = calloc(len+1, 1);
9    for (int i = 1; i <= len; ++i)
10     rev[len-i] = str[i-1];
11   return rev;
12 }
13
14 int main(void) {
15   puts(reverse(":)"));
16   puts(reverse(NULL));
17 }
```

Figure 1: An example program.

In general, the programmer does not want to be warned about every possible error, all the time; rather, they insert annotations into the code to mark things about which they would like to receive static warnings. Our approach in gradual program analysis is to interpret an actual annotation as requesting "pessimistic uncertainty" from the analysis (give static warnings) and interpret the lack of an annotation as requesting "optimistic uncertainty" (don't give static warnings, but do insert checks to catch null-pointer bugs at runtime).

Our framework is general, applying not just to null-pointer analysis but to a broad class (defined more precisely in Section 2) of static analyses that are based on abstract interpretation [5]. We use techniques from Abstracting Gradual Typing (AGT) [8] to give a mechanistic procedure that starts with a static analysis which assumes complete annotations everywhere, and produces a conservative extension of that analysis which can handle missing annotations anywhere. This reduces false positives (see section 3 for

$$
\begin{array}{rcl}
x, y, z & \in & \textsc{Var} \\
a, b & \in & \textsc{Ann} \\
m & \in & \textsc{Proc} \\
n & \in & \textsc{Val} = \mathbb{N} \\
\textsc{Inst} & ::= & (\texttt{const}, x, n) \mid (\texttt{assign}, x, y) \mid (\texttt{if}, x) \mid (\texttt{else}, x) \\
& \mid & (\texttt{proc}, m, x, a) \mid (\texttt{return}, a, x) \mid (\texttt{call}, x, m, a, y, b) \\
& \mid & (\texttt{alloc}, x, n) \mid (\texttt{load}, x, y) \mid (\texttt{store}, x, y) \\
& \mid & (\texttt{add}, x, y, z) \mid (\texttt{subtract}, x, y, z)
\end{array}
$$

Figure 2: Syntax for the instruction set Inst in our running example.

more details on this point) by allowing the programmer to specify where they would and would not like to receive static warnings about potential issues in their code. Our analysis framework is also sound: that is, if the initial static analysis is sound, then the resulting gradual analysis will also be sound, in the sense that it will insert runtime checks to catch any errors that are not reported statically. Finally, gradual program analysis is truly "gradual" in the sense that it satisfies the gradual guarantee [11], so the programmer can reason about how a gradual analysis system will respond to the addition or removal of annotations in their codebase: in essence, removing annotations doesn't cause things to break.

Section 2 discusses our formal system. Then Section 3 discusses a prototype that we built as a Facebook Infer checker to explore the application of our general framework to the specific context of Java null-pointer analysis with `@Nullable` and `@NonNull` annotations. Finally, Section 4 gives a recap and an overview of future work in this project.

2 FORMALISM

Since our framework does not assume any specific underlying language, subsections 2.1 and 2.2 describe the restrictions we must impose on that otherwise arbitrary language. Similarly, subsection 2.3 describes what general form the initial static analysis must take. Then subsection 2.4 gives the method for transforming this static analysis into a gradual analysis, and subsection 2.5 shows how our framework inserts runtime checks to retain soundness. Subsection 2.6 lists some desirable properties that our formal system satisfies. Throughout this whole section we will continue the running example from the introduction, with each such example paragraph set apart from the main text by indentation and prefaced with the italicized word *Example*.

2.1 Language Syntax

Each program p is represented by its control-flow graph. There is a set of opcodes Code, and each node in the graph contains one instruction from the set Inst of the form

$$(c, o_1, o_2, \dots, o_{n_c}) \quad \text{where} \quad c \in \textsc{Code}.$$

The set from which each operand o_i comes depends both on the opcode and the operand's position i in the list.

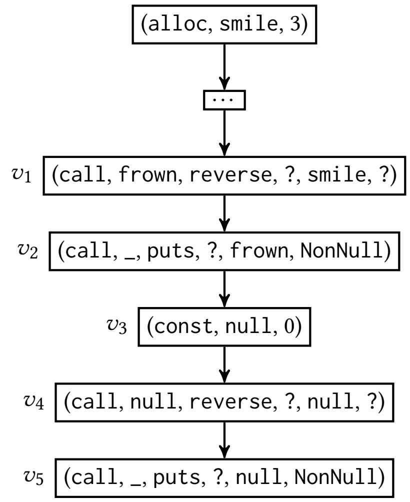

Figure 3: Partial translation of `main`.

> *Example.* Figure 2 shows the syntax for instructions in the control-flow graph language we shall use in this running example. Our set of opcodes is $\textsc{Code} = \{\texttt{const}, \texttt{assign}, \dots\}$, and Proc is the set of procedure names. The other sets Var and Ann are described below. Note that it is not generally the case that $\textsc{Val} = \mathbb{N}$ in our broader framework.

One of the sets from which an operand can come is the set of local variable names Var. At runtime, local variables can take on values from the set Val; that is, one of the components of the runtime state is an environment $\rho \in \textsc{Env} = \textsc{Var} \rightharpoonup \textsc{Val}$ mapping some variable names to values. Another special operand set is the set of analysis annotations Ann, which comes with a "blank" element $? \in \textsc{Ann}$ that corresponds to omitting the annotation entirely.

> *Example.* Figure 3 shows a partial translation of `main` from Figure 1 into the IR defined in Figure 2.

2.2 Language Runtime Semantics

The language must define a small-step operational semantics $\longrightarrow_p$, which is a binary relation on the set of runtime states, and annotations cannot change the runtime semantics of the program, except by possibly making some behavior undefined. That is, say we take a program p, go through each instruction $(c, o_1, \dots, o_{n_c})$ in p, and replace o_i with ? whenever $o_i \in \textsc{Ann}$, to get a modified program p'. For every pair of states ξ and ξ', if $\xi \longrightarrow_p \xi'$ then $\xi \longrightarrow_{p'} \xi'$.

> *Example.* The presence of `NonNull` annotations on the arguments to `puts` at nodes v_2 and v_5 in Figure 3 can cause the program to "fail early" at those points (before calling `puts` at all), but cannot cause other actual behavior.

2.3 Static Analysis

Next, we assume that we are given a sound static analysis that works on the set of all programs which have no missing annotations. Specifically, we must have a set of abstract values $\textsc{Abst} = \textsc{Ann} \setminus$

$$\begin{tikzcd} & \texttt{Nullable} \arrow[dl, no head] \arrow[dr, no head] & \\ \texttt{Null} & & \texttt{NonNull} \end{tikzcd}$$

Figure 4: The semilattice $\textsc{Abst}$ used by our example static analysis.

$$\begin{aligned} \textsc{flow}((\texttt{assign}, x, y), \sigma) &= \sigma[x \mapsto \sigma(y)] \\ \textsc{flow}((\texttt{if}, x), \sigma) &= \sigma[x \mapsto \texttt{NonNull}] \\ \textsc{flow}((\texttt{call}, x, m, a, y, b), \sigma) &= \sigma[x \mapsto a] \\ \textsc{flow}((\texttt{load}, x, y), \sigma) &= \sigma[x \mapsto \texttt{Nullable}][y \mapsto \texttt{NonNull}] \end{aligned}$$

Figure 5: Part of the flow function used by our example static analysis.

$\{?\}$ which correspond to sets of concrete values via an injective concretization function $\gamma : \textsc{Abst} \to \mathcal{P}(\textsc{Val})$. Our set of abstract values forms a semilattice via the partial order

$$a \sqsubseteq b \quad \iff \quad \gamma(a) \subseteq \gamma(b),$$

which in turn induces a join function $\sqcup : \textsc{Abst} \times \textsc{Abst} \to \textsc{Abst}$. The analysis must also come with a flow function

$$\begin{aligned} &\textsc{flow} : \textsc{Inst}' \times \textsc{Map} \to \textsc{Map} \\ &\quad \text{where} \quad \textsc{Inst}' \text{ is all instructions without ?} \\ &\quad \text{and} \quad \textsc{Map} = \textsc{Var} \rightharpoonup \textsc{Abst}. \end{aligned}$$

This flow function must be monotonic (in the second parameter) and locally sound, so it serves as an accurate approximation of the concrete semantics.

Example. Figure 4 shows the (starting) semilattice we will be using in this running example, where $\gamma(\texttt{Nullable}) = \textsc{Val}$, $\gamma(\texttt{Null}) = \{0\}$, and $\gamma(\texttt{NonNull}) = \textsc{Val} \setminus \{0\}$. Part of our flow function is shown in Figure 5.

The static analysis uses these local pieces $\textsc{flow}$ and $\sqcup$ to obtain a global table of analysis results by running the Kildall fixpoint algorithm [10] on the control flow graph of the program. For each node in the control flow graph (holding instruction $\iota \in \textsc{Inst}'$), we end up with a map $\sigma \in \textsc{Map}$ that describes all the possible concrete states that the program could be in at this node. The static analysis must supply a safety function $\textsc{safe} : \textsc{Inst}' \times \textsc{Var} \to \textsc{Abst}$ which serves to specify the set of "safe values" for each variable at each instruction. If our analysis results tell us that $\sigma(x) \sqsubseteq \textsc{safe}(\iota, x)$ for all $x \in \textsc{Var}$ in the domain of σ, and this holds at all nodes in the program control flow graph, then the static analysis deems the program to be "valid."

Example. Since our current static analysis requires full annotations, we must replace every ? in Figure 3 with a `Nullable` annotation before we run the analysis. After we do this, we obtain Table 1. Each row shows the $\sigma \in \textsc{Map}$ that describes all possible runtime states right before the instruction at that node is run. For instance, at node v_2, we have

Table 1: Part of the results from the static analysis fixpoint.

	`smile`	`frown`	`null`
v_1	NonNull		
v_2	NonNull	Nullable	
v_3	NonNull	Nullable	
v_4	NonNull	Nullable	Null
v_5	NonNull	Nullable	Nullable

$$\begin{aligned} \textsc{safe}((\texttt{return}, a, x), x) &= a \\ \textsc{safe}((\texttt{call}, x, m, a, y, b), y) &= b \\ \textsc{safe}((\texttt{load}, x, y), y) &= \texttt{NonNull} \\ \textsc{safe}((\texttt{store}, x, y), x) &= \texttt{NonNull} \end{aligned}$$

Figure 6: Part of the safety function used by our example static analysis.

$\sigma(\texttt{frown}) = \texttt{Nullable}$, but if we let ι be the instruction at v_2 then $\textsc{safe}(\iota, \texttt{frown}) = \texttt{NonNull}$ according to Figure 6; thus, the analysis does not deem our program valid. This particular warning, though, is a false positive.

By construction, this static analysis satisfies a soundness property: if the analysis deems a program "valid" (that is, if the analysis gives no static warnings), then the program must continue to step until it reaches a predesignated termination point.

2.4 Gradual Analysis

Similar to how the first step in Abstracting Gradual Typing (AGT) [8] is to extend the set of types, our first step in gradual analysis is extend the semilattice $\textsc{Abst}$ to form a larger semilattice $\widetilde{\textsc{Abst}} \supseteq \textsc{Abst}$, to be defined more concretely below. Following AGT, we need an injective function $\widetilde{\gamma} : \widetilde{\textsc{Abst}} \to \mathcal{P}^+(\textsc{Abst})$ to give meaning to these new gradual abstract elements. We then define $\widetilde{\gamma}(a) = \{a\}$ for all $a \in \textsc{Abst}$. We also insist that $? \in \widetilde{\textsc{Abst}}$ and let $\widetilde{\gamma}(?) = \textsc{Abst}$. Given $\widetilde{\gamma}$, we can now lift predicates on $\textsc{Abst}$ (such as the order relation $\sqsubseteq$) to predicates on $\widetilde{\textsc{Abst}}$. For $\widetilde{a}, \widetilde{b} \in \widetilde{\textsc{Abst}}$:

$$\widetilde{a} \mathrel{\widetilde{\sqsubseteq}} \widetilde{b} \quad \iff \quad a \sqsubseteq b \text{ for some } a \in \widetilde{\gamma}(\widetilde{a}) \text{ and } b \in \widetilde{\gamma}(\widetilde{b})$$

Observe that for any $a \in \textsc{Abst}$ we have $? \mathrel{\widetilde{\sqsubseteq}} a \mathrel{\widetilde{\sqsubseteq}} ?$, so $\widetilde{\sqsubseteq}$ is not a partial order. Thus it cannot induce a join operation with the properties we need for computing a fixpoint. To get such a join, we need a bit more structure that will allow us to lift functions.

For $\widetilde{a}, \widetilde{b} \in \widetilde{\textsc{Abst}}$, we can easily construct the set

$$\{a \sqcup b : a \in \widetilde{\gamma}(\widetilde{A}) \text{ and } b \in \widetilde{\gamma}(\widetilde{b})\} \in \mathcal{P}^+(\textsc{Abst}).$$

Thus we need a function $\widetilde{\alpha} : \mathcal{P}^+(\textsc{Abst}) \to \widetilde{\textsc{Abst}}$. In AGT, this forms a Galois connection with $\widetilde{\gamma}$, so the following must hold:

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

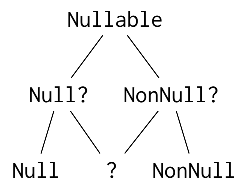

Figure 7: The semilattice structure induced by the lifted join $\widetilde{\sqcup}$ on our running example. Specifically, this is the Hasse diagram of the partial order $\{(a, b) : a \mathbin{\widetilde{\sqcup}} b = b\}$.

It can be shown that if $\widetilde{\alpha}$ exists, it must be unique. Then if $\widetilde{\alpha}$ exists, we can define the lifted join

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

to use in the fixpoint algorithm.

Example. As it turns out, it is insufficient to let $\widetilde{\textsc{Abst}} = \textsc{Abst} \cup \{?\} = \textsc{Ann}$. If we do this for the semilattice we have been using in our running example (shown in Figure 4), we end up with

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

This means that our lifted join would not be associative, preventing the existence of a unique fixpoint.

So far, we have not yet precisely defined $\widetilde{\textsc{Abst}}$. To satisfy all the properties we need for our lifted join function, we now choose

$$\widetilde{\textsc{Abst}} = \textsc{Abst} \cup \{?\} \cup \{a? : a \in \textsc{Abst}\}$$

where $\widetilde{\gamma}(a?) = \{b \in \textsc{Abst} : a \sqsubseteq b\}$ for $a \in \textsc{Abst}$. Importantly, it can be shown that if $(\textsc{Abst}, \sqcup)$ has finite height h—which it must, in order to ensure that the fixpoint algorithm terminates in a finite amount of time—then $(\widetilde{\textsc{Abst}}, \widetilde{\sqcup})$ is a semilattice with finite height $h + 1$. Notice that our $a?$ is different from, for instance, the $\phi \wedge ?$ that appears in gradual verification [2], since the latter means "ϕ or anything more specific than it," while the former means "a or anything less specific than it."

Example. Figure 7 shows the lifting of the semilattice from Figure 4, with its join function $\widetilde{\sqcup}$. (The lifted order relation $\widetilde{\sqsubseteq}$ is not shown.) Notice that since $\widetilde{\gamma}$ is injective, we have `Nullable? = Nullable` because

$$\begin{aligned} \widetilde{\gamma}(\texttt{Nullable?}) &= \{a \in \textsc{Abst} : \texttt{Nullable} \sqsubseteq a\} \\ &= \{\texttt{Nullable}\} = \widetilde{\gamma}(\texttt{Nullable}). \end{aligned}$$

Now that we have lifted the semilattice, the rest of the gradual analysis system is fairly straightforward. We next need to lift $\textsc{flow}$ to $\widetilde{\textsc{flow}} : \textsc{Inst} \times \widetilde{\textsc{Map}} \to \widetilde{\textsc{Map}}$ (where $\widetilde{\textsc{Map}} = \textsc{Var} \rightharpoonup \widetilde{\textsc{Abst}}$), and also lift $\textsc{safe}$ to $\widetilde{\textsc{safe}} : \textsc{Inst} \times \textsc{Var} \to \widetilde{\textsc{Abst}}$, using the same function lifting technique from AGT.

Table 2: Part of the results from the gradual analysis fixpoint.

	`smile`	`frown`	`null`
v_1	NonNull		
v_2	NonNull	?	
v_3	NonNull	?	
v_4	NonNull	?	Null
v_5	NonNull	?	?

Example. Conveniently, in our running example, the rules for calculating values of $\widetilde{\textsc{flow}}$ and $\widetilde{\textsc{safe}}$ are exactly the same as the rules shown in Figure 5 and Figure 6.

Since we now have a lifted flow function $\widetilde{\textsc{flow}}$ (which can be shown to be monotonic) and join function $\widetilde{\sqcup}$ (which can be shown to satisfy all the properties for the join of a finite-height semilattice), we can simply take any program (possibly with missing annotations) and run it through the exact same fixpoint algorithm as before, swapping out the "subroutines" $\textsc{flow}$ and $\sqcup$ for $\widetilde{\textsc{flow}}$ and $\widetilde{\sqcup}$, respectively. Again we end up with a $\widetilde{\sigma} \in \widetilde{\textsc{Map}}$ for each node in the control flow graph (holding instruction $\iota \in \textsc{Inst}$), so we check whether $\widetilde{\sigma}(x) \mathrel{\widetilde{\sqsubseteq}} \widetilde{\textsc{safe}}(\iota, x)$ for all $x \in \textsc{Var}$. If this check fails anywhere, we emit a static warning. If not, then our gradual analysis has deemed the program to be "statically valid."

Example. When we run our gradual analysis on Figure 3, we obtain Table 2. At node v_2, with instruction ι, we again have $\widetilde{\textsc{safe}}(\iota, \texttt{frown}) = \texttt{NonNull}$, but now we also have $\widetilde{\sigma}(\texttt{frown}) = ?$, and $? \mathrel{\widetilde{\sqsubseteq}} \texttt{NonNull}$. Thus, our gradual analysis does not give a false positive warning at v_2. However, it similarly does not give a warning at v_5, which actually will yield a null-pointer dereference at runtime.

2.5 Runtime Checks

The final step, assuming that our program p is "statically valid" according to the gradual analysis, is to insert runtime checks in places where the analysis might have been too optimistic. That is, we construct a new program p' by finding all points with $\widetilde{a} = \widetilde{\sigma}(x)$ and $\widetilde{b} = \widetilde{\textsc{safe}}(\iota, x)$ such that

$$a \not\sqsubseteq \bigsqcup \widetilde{\gamma}(\widetilde{b}) \text{ for some } a \in \widetilde{\gamma}(\widetilde{a});$$

at each of these points, the semantics of p' check to ensure that the runtime value of x lies within $\bigsqcup \widetilde{\gamma}(\widetilde{b})$, and if not, the program steps into a dedicated error state. Otherwise, the semantics of p' are the same as those of p.

Example. According to Table 2, we should insert a runtime check for `frown` being `NonNull` at v_2, and for `null` being `NonNull` at v_5.

2.6 Properties

Our formalism has several nice properties:

(1) It is a *conservative extension* of the original analysis.
(2) It is *sound* if the original analysis is sound.

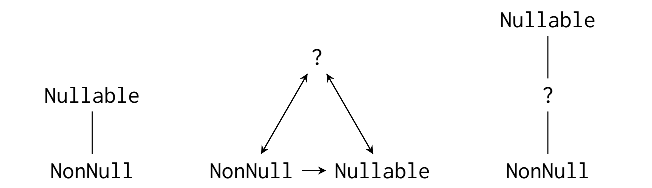

Figure 8: *Left:* The original null-pointer semilattice. *Middle:* The lifted semilattice ordering, where each directed edge $a \to b$ means $a \mathrel{\widetilde{\sqsubseteq}} b$. (Self-loops are omitted.) *Right:* The semilattice structure induced by the lifted join $\widetilde{\sqcup}$.

(3) It satisfies the *gradual guarantee* [11]: if you remove annotations from a program that currently is statically valid or one that succeeds at runtime, then the resulting program will also be statically valid or succeed at runtime, respectively.

We have formally proven the first property, along with all the properties of the semilattice lifting discussed earlier. We have not yet formally proven the second and third properties, but we have detailed sketches of proofs for them.

3 PROTOTYPE

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,"[1] based on the development presented above. Our prototype also uses a slightly simpler starting lattice, shown in Figure 8. Also note that because Infer only supports analysis and not modification of the program being analyzed, our prototype only returns a list of runtime checks to insert, rather than actually inserting those checks.

To evaluate this prototype, we used the 18 repositories which Uber used to evaluate their NullAway analysis tool [3]. We ran Facebook Infer's existing Eradicate and Nullsafe checkers 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. After examining all of these 2371 warnings, we found only 57 that could 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 statically-reported false positives than Facebook's existing null-pointer analysis tools. However, experiments should be performed on repositories with more defects, to see how often our framework catches bugs statically and how often those bugs are only caught at runtime.

4 CONCLUSION

We have given motivation and noted precedent for program analyses that have separate concepts for pessimistic and optimistic uncertainty, and have described a formal system for constructing such analyses via optional annotations and the Abstracting Gradual Typing methodology [8]. We have also listed some desirable formal properties that our system possesses, and discussed some initial empirical evaluation that we have performed using a prototype of our formal system.

Future work will involve completing the formal proofs that our framework is sound and satisfies the gradual guarantee, implementing a prototype that implements our entire general framework rather than just a single analysis, and conducting robust experiments to evaluate such a prototype.

[1] https://github.com/gradual-verification/graduator

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.

Jenna Wise and Jonathan Aldrich from Carnegie Mellon University provided invaluable mentorship throughout this entire research project, which began in summer 2019. Also, several other people made important contributions to this work, including Éric Tanter from the University of Chile, Johannes Bader from Facebook, and Joshua Sunshine from Carnegie Mellon University. An overview of this research was published as the author's undergraduate thesis; Ethan Smith and Daniel Majcherek from Liberty University generously oversaw the writing and editing of that thesis work.

REFERENCES

[1] Nathaniel Ayewah and William Pugh. 2010. The google findbugs fixit. In *Proceedings of the 19th international symposium on Software testing and analysis*. 241–252.
[2] Johannes Bader, Jonathan Aldrich, and Éric Tanter. 2018. Gradual Program Verification. In *International Conference on Verification, Model Checking, and Abstract Interpretation*. Springer, 25–46.
[3] Subarno Banerjee, Lazaro Clapp, and Manu Sridharan. 2019. NullAway: Practical Type-Based Null Safety for Java. *arXiv preprint arXiv:1907.02127* (2019).
[4] Mike Barnett, Manuel Fahndrich, Francesco Logozzo, and Diego Garbervetsky. 2007. Annotations for (more) precise points-to analysis. (2007).
[5] 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.
[6] 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* (Athens, Greece) *(SPLASH Companion 2019)*. ACM, New York, NY, USA, 52–53. https://doi.org/10.1145/3359061.3361082
[7] Samuel Estep, Jenna Wise, Jonathan Aldrich, Éric Tanter, Johannes Bader, and Joshua Sunshine. 2020. *Gradual Program Analysis*. https://popl20.sigplan.org/details/wgt-2020-papers/9/Gradual-Program-Analysis
[8] 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* (St. Petersburg, FL, USA) *(POPL '16)*. ACM, New York, NY, USA, 429–442. https://doi.org/10.1145/2837614.2837670
[9] 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.
[10] 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.
[11] 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.
[12] Fausto Spoto. 2011. Precise null-pointer analysis. *Software & Systems Modeling* 10, 2 (2011), 219–252.