On quantifying the degree of unsoundness of static analyses

Dimitrios Vardoulakis
Google
dimvar@google.com

Introduction

The theory and practice of static analysis differ greatly. In static-analysis research, the focus is commonly on analyses that are proven sound, apply to small calculi or toy languages, and use sophisticated techniques to achieve precision. In industry, static analyses are unsound, apply to real languages, and use simple techniques in order to scale to large programs [3].

Static-analysis researchers avoid unsound analyses, because it is not clear what one can formally say about an unsound analysis. Sure, it helps people in practice, but what precise statement can one make about it? Is the merit of an unsound analysis based purely on opinion?

At Google, I work on Closure Compiler, a type checker and optimizer for JavaScript. It is used to optimize all of Google's major frontends. Closure Compiler uses an unsound type system [5], but despite that, it uses the types to perform optimizations. When I mention this to static-analysis researchers in academia, they find it hard to believe. We have used unsound types for optimization for years, and are confident in the approach, but can we find a formal way to explain why the types are "correct in practice"?

I believe it is beneficial to characterize unsound analyses mathematically; to develop rigorous ways of evaluating them and formal models describing their behavior. Then, theoreticians would be able to approach unsound analyses systematically, and develop novel ways of reasoning about such analyses. In addition, practitioners would be able to compare the pros and cons of various analyses in a precise way.

In a recent talk [5], I proposed using ideas from machine learning to evaluate static analyses. Consider a classifier that decides whether an email is spam. The classifier can err in two directions: it can say that a non-spam email is spam (false positive), or that a spam email is not spam (false negative). The fact that the classifier has false negatives does not make it useless. We should think of an unsound analysis in the same way. We cannot prove that the analysis eliminates certain runtime errors, but we can evaluate it based on precision and recall. Hicks makes similar suggestions in a recent blog post [2].

The goal of this paper is to expand on these ideas and encourage others to look at unsound analyses in a rigorous way.

My running example of an unsound analysis will be type checking of JavaScript. There are currently three industrial-strength type checkers for JavaScript, and they are all unsound (Closure Compiler, TypeScript, and Flow).

Using a labeled dataset to evaluate static analyses

ImageNet is a public dataset used to evaluate image-recognition systems. It contains millions of images, separated into thousands of categories. A classifier must look at an image in the dataset and label it as belonging to one of the categories.

It is possible to create such a dataset (albeit at a smaller scale initially) to evaluate JavaScript type checkers. For example, the type checker of Closure Compiler categorizes warnings into about a hundred different diagnostic types, such as: invalid operand type, invalid argument type, wrong argument count, inheritance cycle, *etc*. The test suite contains about 2400 unit tests, and for each unit test we know the diagnostic types of the expected warnings. One can standardize the diagnostic types across all JavaScript type checkers, and use test suites to seed a curated dataset.

To evaluate a type checker on the dataset, we run it on all programs in the dataset and, for each program, check the warnings it produces against the expected warnings. Then, we can measure:

- How many times it produces an expected warning (true positive).
- How many times it produces an unexpected warning (false positive).
- How many times it fails to produce an expected warning (false negative).
- How many times it correctly produces no warning (true negative).

This approach has several benefits. First, we have an objective way of comparing type checkers.

Second, we can treat the type checker as a black box; we only care about its effectiveness, not about the specifics of the algorithm it uses. Formalizing the type-checking algorithm itself, *e.g.*, as a collection of type rules, would be much more difficult due to the complexity of JavaScript.

Third, a sound type checker is no longer special; it becomes just another point in the design spectrum, and can be evaluated on the same dataset. In fact, we would likely find that a sound type checker is not a good choice for JavaScript. To achieve soundness, we must analyze the dynamic parts of the language conservatively, which results in overapproximating the types of many expressions, which results in many false positives and low precision. Machine-learning systems also behave this way: increasing recall results in low precision, and increasing precision results in low recall [1].

Using an instrumented interpreter as the source of truth

Using a labeled dataset is good for evaluating static analyses for full-featured languages. What can we do if we are working on an analysis for a small, experimental language L, where a dataset is not available?

One option is to use an instrumented interpreter for L as the source of truth. To find the effectiveness of the analysis on a program P, we can compare the analysis result to the result of executing P using the interpreter. Without loss of generality, we assume that P does not take any input. This assumption lets us focus on the single execution of P, rather than the set of executions under all possible inputs.

We define an abstraction function f from concrete values (values computed at runtime) to abstract values (values computed during static analysis). We give each expression in P a unique label, interpret the program, collect all values for each label, and apply f to get a set of abstract values for each label. Then, we run our analysis on P and calculate a different set of abstract values for each label.

For some particular label, let S be the set computed by the interpreter, and S' the set computed by the static analysis.

- If S is equal to S', the analysis computed the correct result at that label.
- If S is a subset of S', the analysis overapproximated and we have a false positive.
- If S' is a subset of S, the analysis underapproximated and we have a false negative.
- Else, the analysis result is neither an overapproximation nor an underapproximation.

In this manner, we can compute the percentage of correct results, false positives, and false negatives for P.

To estimate the effectiveness of the analysis on the whole language L, we can follow a Monte Carlo approach and compare the analysis results and the runtime results for a large number of randomly generated programs. As with a labeled dataset, the instrumented-interpreter approach can be used to evaluate both sound and unsound analyses.

It is worth mentioning that we can use familiar techniques from the design of sound analyses for unsound analyses as well. For example, we can express an unsound analysis as an abstract interpretation using operational semantics, in the style of [4]. But instead of proving a simulation theorem, we use an instrumented interpreter to get a "soundness percentage" for our analysis.

Conclusion

Machine learning researchers have long used measures such as precision and recall to score systems that have both false positives and false negatives. Similar measures exist for evaluating medical tests: sensitivity and specificity. I believe that such measures are a natural fit for evaluating static analyses as well.

When studying static analysis, we learn that soundness is a binary property; either an analysis is sound or not. But static-analysis practitioners view soundness as a spectrum, and trade off "some" soundness in exchange for precision and scalability. They view program analysis through a statistical lens instead of a logical lens. In this paper, I propose ways to formalize this intuition, so that unsound analyses can be studied in a mathematically rigorous way.

References

[1] Trading off precision and recall. Coursera course on Machine Learning, https://www.coursera.org/learn/machine-learning/lecture/CuONQ/trading-off-precision-and-recall.

[2] Michael Hicks. What is soundness (in static analysis)?, October 2017. http://www.pl-enthusiast.net/2017/10/23/what-is-soundness-in-static-analysis/.

[3] Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondrej Lhoták, José Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z. Guyer, Uday P. Khedker, Anders Møller, and Dimitrios Vardoulakis. In defense of soundiness: a manifesto. *Communications of the ACM*, 58(2):44–46, 2015.

[4] Matthew Might. *Environment Analysis of Higher-Order Languages*. PhD thesis, Georgia Institute of Technology, 2007.

[5] Dimitrios Vardoulakis. The design of JavaScript type systems. In *Strange Loop*, September 2017.