PL Perspectives

Perspectives on computing and technology from and for those with an interest in programming languages.

What Does It Mean for a Program Analysis to Be Sound?

by Ilya Sergey on Aug 7, 2019 | Tags: abstract interpretation, concurrency, dynamic analysis, soundness, static analysis, testing

Alice and Bob work as software engineers in a company with a large code base. One day, their manager realizes that the company's main software application can be executed far more efficiently if it is made concurrent. So she proposes to redesign some of its core components. Alice and Bob are tasked to

CONTRIBUTE

Editor: Michael Hicks

Contribute to the Blog

RECENT POSTS

Can Journals Make PL More Accessible?

What Does It Mean for a Program Analysis to Be Sound?

Program Synthesis in 2019

Towards a Universal Quantum Programming Language

ACM Conferences and the Cost of Carbon

ARCHIVES

implement tools that detect concurrency bugs in code, to help their colleagues avoid mistakes during the refactoring. Their focus is on *data races*, which occur when multiple threads try to access a single memory location in a conflicting way.

Alice and Bob decide to try two different approaches to a program analysis. Alice adopts a *dynamic* approach inspired by tools such as FastTrack and Narada. Alice's approach instruments code to detect data races at run-time, using automatically generated inputs to test the instrumented code. The approach requires almost no input from developers, and immediately helps to detect a number of newly introduced bugs in the refactored components. However, it also fails to catch a few races which slip into production.

Bob takes a *static* approach to the program analysis. Programmers must provide a few annotations indicating their intent to introduce concurrency in the code, following the ideas of tools such as Chord and Checker Framework. Bob's analysis automatically analyzes the annotated code and reports many potential races. A number of legitimate bugs are detected by Bob's analysis. But the developers are unhappy about the time they wasted looking at fragments of code that Bob's analysis incorrectly marked as problematic.

In sum, both tools find bugs. But Alice's tool misses some bugs and Bob's tool sometimes complains about non-bugs. Despite these differences, both Alice and Bob claim that that their tools are *sound*, that is, they always catch the bugs in a correct way. Can they both be right? The answer is 'yes', and both approaches to software validation can be adopted, but they have different goals in mind: testing and verification.

Soundness in Dynamic and Static Analyses

A good testing framework finds important failures. It allows one to argue that a failed test *always* indicates a bug, giving no *false positives*. In the dynamic analysis community this quality is referred to as *soundness*. For instance, a sound dynamic data race detector reports only true races.

A traditional goal of static analyses is to argue about the total absence of run-time errors. It is, however, well-known that implementing a procedure for ensuring the absence of both false positives (reported non-bugs) and false negatives (missed bugs) with regard to any "interesting" program property (e.g., presence of data races) is impossible. To circumvent this limitation, Cousot and Cousot, in their seminal 1977 paper, introduced *abstract interpretation* as a way to define static program analyses as procedures for computing "over-approximations" of finite property-specific abstractions of *all* program runtime executions. An abstract interpretation-based analysis can be used as a program verifier: if it reports *no* bugs, there should be no corresponding run-time property violations in any of the program's executions. In the static analysis community, this quality (the absence of *false negatives*) is also, quite confusingly, referred to as *soundness*.

We can now see that soundness in the context of dynamic and static program analyses indeed means two different (indeed, opposite) things.

When Static Analysis is More Like Testing

Any compiler for a statically-typed language features a very specific instance of a static analysis that soundly prevents a large class of runtime errors, known as *type systems*. Static analyses are also instrumental for enabling various compile-time optimisations. The soundness of static analyses included into a compiler pipeline is, thus, a must-requirement for preserving the semantics of the program being compiled, and, thus, for guaranteeing the correctness of the compiler.

At the same time, many static analyses used in practice to detect bugs and enable code refactorings are close in spirit to testing frameworks. For instance, most of the major IDEs for Java, such as IntelliJ IDEA and Eclipse offer large suites of "code inspections" aimed to detect various programming mistakes statically. For the sake of a better user experience and faster response, those tools deliberately make choices that render the underlying analyses unsound. This phenomenon has been recognised by the programming languages academic community, which studied the impact of certain language features to soundness of static analyses, and coined the term *soundiness* for indicating a static analysis that is "almost sound".

The following question naturally arises: if a static analysis is used as a testing tool and is known to be unsound in a static analysis (verification) sense, can it be shown to be sound in the dynamic analysis (testing) sense?

A Soundness Theorem for an Unsound Static Analysis

The story in the beginning of this article has been adapted from a real case study that led to the development of the Facebook RacerD static analyzer for Java while migrating the Litho library for UI rendering from a single-threaded to a multi-threaded model. RacerD aimed to complement existing tools for dynamic detection of data races, such as RacerX and FastTrack. It has proven to be a very efficient tool for catching data races in refactored concurrent code. My co-authors and myself were puzzled by the fact that, while being unsound as a static analysis, RacerD was very close to being sound as a testing tool: as we report in our OOPSLA 2018 paper, it almost never reported a false race. In other words, it had almost no false positives.

The True Positives Theorem

Fascinated by this observation, we decided to abandon our initial understanding of RacerD soundness, inspired by abstract interpretation. Instead, we focused on proving RacerD to be *sound as a testing framework*. The result, which we called *A True Positives Theorem* is described in our recent POPL 2019 paper. The paper presents, to the best of our knowledge, the first proof that a static analysis is sound in a dynamic sense, despite being (or rather, because it is) deliberately unsound in a static sense.

Establishing this result was not straightforward. Specifically, in order to prove the desired theorem with regard to the "standard" semantics of Java and keep the analysis scalable at the same time, we would have to make it treat control- and data-flow conservatively. This would lead to a significant loss of precision: while not reporting false positives, such an analysis would also fail to report many true positives. Therefore, our subject analysis assumes a *non-deterministic* program semantics, in which every conditional execution branch and a body of a while-loop is executed at least once. For instance, if a race were introduced by a code guarded by a conditional statement, it would be reported by our analysis and considered a true one according to the statement of the theorem. This seemingly controversial decision has been justified by our practical observations: it is rare for "dead code" to occur in critical parts of real-life software, hence we considered any code as "potentially executable", and, therefore, capable of introducing data races.

Conclusion

Historically, static analyses are seen as procedures for computing "over-approximations" of runtime program behaviors, while dynamic analyses targeted the “under-approximation”. These two views shaped the perceived notion of the analysis soundness in the corresponding settings. In practice, however, static analyses are frequently used as tools for bug detection rather than verification, which means that they require a different notion of soundness and the corresponding theorem. The primary aim of such a theorem is, thus, not only provide formal guarantees to the programmers, but also clarify to the analysis designers the nature and the purpose of the analysis algorithm they develop.

I believe that our True Positives Theorem result paves the way to a uniform formal treatment of soundness in both testing and verification. That is, we might hope to have in the future a methodology for designing analyses that combine the traits of both approaches, coming with the corresponding notions of correctness. It is also likely that the already existing techniques for reducing the numbers of false positives in verification-style analyses, such as pre-analysis or Bayesian inference, can be employed to increase the numbers of true positives in testing-style analyses.

Bio: *Ilya Sergey is a tenure-track Associate Professor at Yale-NUS College and the School of Computing of National University of Singapore. He does research in programming language theory, semantics, software verification, and program synthesis. He was awarded the 2019 AITO Dahl-Nygaard Junior Prize for his contributions in the development and application of programming language techniques to various problems across the programming spectrum.*

those of ACM SIGPLAN or its parent organization, ACM.