In Defense of Soundiness: A Manifesto

Ben Livshits, Manu Sridharan, Yannis Smaragdakis, Ondřej Lhoták, J. Nelson Amaral, Bor-Yuh Evan Chang, Sam Guyer, Uday Khedker, Anders Møller, and Dimitrios Vardoulakis

Microsoft Research, Samsung Research America, University of Athens, University of Waterloo, University of Alberta, University of Colorado Boulder, Tufts University, IIT Bombay, Aarhus University, Google

Static program analysis is a key component of many software development tools, including compilers, development environments, and verification tools. Practical applications of static analysis have grown in recent years to include tools by companies such as Coverity, Fortify, GrammaTech, IBM, and others. Analyses are often expected to be *sound* in that their result models all possible executions of the program under analysis. Soundness implies that the analysis computes an over-approximation in order to stay tractable; the analysis result will also model behaviors that do not actually occur in any program execution. The *precision* of an analysis is the degree to which it avoids such spurious results. Users expect analyses to be sound as a matter of course, and desire analyses to be as precise as possible, while being able to *scale* to large programs.

Soundness would seem essential for any kind of static program analysis. Soundness is also widely emphasized in the academic literature. Yet, in practice, soundness is commonly eschewed: we are not aware of a *single* realistic whole-program[1] analysis tool (*e.g.*, tools widely used for bug detection, refactoring assistance, programming automation, *etc.*) that does not purposely make unsound choices. Similarly, virtually all published whole-program analyses are unsound and omit conservative handling of common language features when applied to *real programming languages.*

The typical reasons for such choices are engineering compromises: implementers of such tools are well aware of how they could handle complex language features soundly (*e.g.*, by assuming that a complex language feature can exhibit *any* behavior), but do not do so because this would make the analysis *unscalable* or *imprecise* to the point of being useless. Therefore, the dominant practice is one of treating soundness as an engineering choice.

In all, we are faced with a paradox: on the one hand we have the ubiquity of unsoundness in any practical whole-program analysis tool that has a claim to precision and scalability; on the other, we have a research community that, outside a small group of experts, is oblivious to any unsoundness, let alone its preponderance in practice.

Our observation is that the paradox can be reconciled. The state of the art in realistic analyses exhibits consistent traits, while also integrating a sharp discontinuity. On the one hand, typical realistic analysis implementations have a *sound core*: most common language features are *over-approximated*, modeling all their possible behaviors. Every time there are multiple options (*e.g.*, branches of a conditional statement, multiple data flows) the analysis models all of them. On the other hand, some specific language features, well known to experts in the area, are best *under-approximated*. Effectively, every analysis pretends that perfectly possible behaviors cannot happen. For instance, it is conventional for an otherwise sound static analysis to treat highly-dynamic language constructs, such as Java reflection or *eval* in JavaScript, under-approximately. A practical analysis, therefore, may pretend that *eval* does nothing, unless it can precisely resolve its string argument at compile time.

We introduce the term *soundy* for such analyses. The concept of *soundiness* attempts to capture the balance, prevalent in practice, of over-approximated handling of most language features, yet deliberately under-approximated handling of a feature subset well recognized by experts. Soundiness is in fact what is meant in many papers that claim to describe a sound analysis. A *soundy* analysis aims to be as sound as possible without excessively compromising precision and/or scalability.

Our message here is threefold:

1. We bring forward the ubiquity of, and engineering need for, unsoundness in the static program analysis practice. For static analysis researchers, this may come as no surprise. For the rest of the community, which expects to use analyses as a black box, this unsoundness is less understood.

[1] We draw a distinction between whole program analyses, which need to model shared data, such as the heap, and modular analyses--e.g., type systems. Although this space is a continuum, the distinction is typically well-understood.

2. We draw a distinction between analyses that are soundy---mostly sound, with specific, well-identified unsound choices---and analyses that do not concern themselves with soundness.
3. We issue a call to the community to identify clearly the nature and extent of unsoundness in static analyses. Currently, in published papers, sources of unsoundness often lurk in the shadows, with caveats only mentioned in an off-hand manner in an implementation or evaluation section. This can lead a casual reader to erroneously conclude that the analysis is sound. Even worse, elided details of how tricky language constructs are handled could have a profound impact on how the paper's results should be interpreted, since an unsound handling could lead to much of the program's behavior being ignored (consider analyzing Eclipse without understanding at least something about reflection; most of the program will likely be omitted from analysis).

Unsoundness: Inevitable and, Perhaps, Desirable?

The typical (published) whole-program analysis extolls its scalability virtues and briefly mentions its soundness caveats. For instance, an analysis for Java will typically mention that reflection is handled "as in past work," while dynamic loading will be (silently) assumed away, as will be any behavior of opaque, non-analyzed code (mainly native code) that may violate the analysis' assumptions. Similar "standard assumptions" hold for other languages. Indeed, many analyses for C and C++ do not support casting into pointers, and most ignore complex features such as `setjmp/longjmp`. For JavaScript the list of caveats grows even longer, to include the `with` construct, dynamically-computed fields (called properties), as well as the notorious `eval` construct.

Can these language features be ignored without significant consequence? Realistically, most of the time the answer is no. These language features are nearly ubiquitous in practice. Assuming the features away excludes the majority of input programs. For example, very few JavaScript programs larger than a certain size omit at least occasional calls to `eval`.

Could all these features be modeled soundly? In principle, yes. In practice, however, we are not aware of a single sound whole-program static analysis tool applicable to industrial-strength programs written in a mainstream language! The reason is that sound modeling of all language features usually destroys the precision of the analysis because such modeling is usually highly over-approximate. Imprecision, in turn, often destroys scalability because analysis techniques end up computing huge results--a typical modern analysis achieves scalability by maintaining precision, thus minimizing the data sets that it manipulates.

Soundness is not even *necessary* for most modern analysis applications, however, as many clients can tolerate unsoundness. Such clients include IDEs (auto-complete systems, code navigation), security analyses, general-purpose bug detectors (as opposed to program verifiers), *etc.* Even automated refactoring tools that perform code transformation are unsound in practice (especially when concurrency is considered), and yet they are still quite useful and implemented in most IDEs. Third-party users of static analysis results---including other research communities, such as software engineering, operating systems, or computer security --- have been highly receptive of program analyses that are unsound, yet useful.

Evaluating Sources of Unsoundness by Language

While an unsound analysis may take arbitrary shortcuts, a soundy analysis that attempts to *do the right thing* faces some formidable challenges. In particular, unsoundness frequently stems from difficult-to-model language features. In the table below we list some of the sources of unsoundness, which we segregate by language.

Language	Examples of commonly ignored features	Consequences of not modeling these features
C/C++	`setjmp/longjmp` ignored	ignores arbitrary side-effects to the program heap
	effects of pointer arithmetic	
	“manufactured” pointers	
Java/C#	Reflection	can render much of the codebase invisible for analysis
	JNI	“invisible” code may create invisible side-effects in programs
JavaScript	eval, dynamic code loading	missing execution
	data flow through the DOM	missing data flow in program

All features listed in the table can have significant consequences on the program, yet are commonly ignored at analysis time. For language features that are most often ignored in unsound analyses (reflection, `setjmp/longjmp`, `eval`, *etc.*), more studies should be published to characterize how extensively these features are used in typical programs and how ignoring these features could affect standard program analysis clients. Recent work analyzes the use of `eval` in JavaScript. However, an informal email and in-person poll of recognized experts in static and runtime analysis failed to pinpoint a single reliable survey of the use of so-called *dangerous* features (pointer arithmetic, unsafe type casts, *etc.*) in C and C++.

Clearly, an improved evaluation methodology is required for these unsound analyses, to increase the comparability of different techniques. Perhaps, benchmarks or regression suites could be assembled to measure the effect of unsoundness. While further work is required to devise such a methodology in full, we believe that, at the least, some effort should be made in experimental evaluations to compare results of an unsound analysis with observable dynamic behaviors of the program. Such empirical evaluation would indicate whether important behaviors are being captured. It really does not help the reader for the analysis' author to declare that their analysis is sound modulo features X and Y, only to discover that these features are present in just about every real-life program! For instance, if a static analysis for JavaScript claims to be "sound modulo eval", a natural question to ask is whether the types of input program this analysis expects do indeed use eval in a way that is highly non-trivial.

Moving Forward

We strongly feel that

- The programming language (PL) research community should embrace soundy analysis techniques and tune its soundness expectations. The notion of soundiness can influence not only tool design but also that of programming languages or type systems. For example, the type system of TypeScript is unsound, yet practically very useful for large-scale development.
- *Soundy is the new sound*; *de facto*, given the research literature of the past decades.
- Papers involving soundy analyses should both explain the general implications of their unsoundness and evaluate the implications for the benchmarks being analyzed.
- As a community, we should provide guidelines on how to write papers involving soundy analysis, perhaps varying per input language, emphasizing which features to consider handling--or not handling.