In Defense of Soundiness

Soundness is frequently proclaimed as a universal virtue in programming language research. When program analysis was primarily used for compiler optimizations, this viewpoint was easy to justify. However, today program analysis is used for a variety of purposes, including bug finding, finding security flaws, suggesting runtime optimization opportunities, etc. In some cases, being sound is valuable, in others, much less so.

Role of Soundness

Our position that *excessive* emphasis on soundness in the programming language community, as represented by publication venues such as PLDI, POPL, FSE, ICSE, SAS, IFCP is misguided. We feel that this exaggerated emphasis that extolls soundness as an undisputed virtue is damaging for several reasons, some well-recognized, others less so:

- Maintaining soundness often leads to other aspects of analysis suffering. A common example of this is program analysis for bug finding, where soundness frequently means to too many false positives, rendering the resulting tool unusable in practice. This may because making conservative assumptions leads to result impression. However, this can also happen because focusing on soundness will lead to difficult to build and fragile less-that-maintainable implementations, sacrificing the chances of such a technique being deployed.
- Interesting and controversial ideas do not make it to these “core PL” conferences, while having little trouble being published elsewhere. This in turn robs the programming languages community of useful inspiration and potentially groundbreaking work and leads to frustration and a sense of “establishment mentality” that is overpowering in defining what kind of work is “legitimate”.

The end result is that the exaggerated focus on soundness, while intellectually challenging and rewarding, leads to a dogmatic perception of research uncommon in other communities such as operating systems and security.

Soundiness

Attacking soundness without providing a different precept would not be very effective. To this end, we propose a concept of *soundiness*, which encompasses a best-effort approach to soundness. Another way to put it may be trying to be sound without bending over backwards to do so.

We argue that soundiness is in fact what is meant in many papers that claim to be sound. For instance, most papers that focus on C and C++ make the assumption that independently-allocated memory regions are not reachable from each other via pointer arithmetic. Similarly, most papers do not offer handling of the setjmp/longjmp construct. When it comes to Java, reflection is a typically the Achilles heel of trying to analyze code statically; it is not handled or handled incompletely in most static analysis efforts. In JavaScript, the list of caveats grows even longer, to include the with construct, dynamically-computed fields (called properties), and eval.

Stating Assumptions

An approach that has been advocated in the literature is to precisely state one’s assumptions. One formula is to declare “The analysis is sound assuming conditions A, B, and C hold about the input program”. While satisfying from the intellectual standpoint, this approach suffers from the following shortcomings:

- Often the assumptions are not too easy to state, yet they invalidate the majority of input programs. For example, saying that “this analysis is sound for all programs that lack reflection” will exclude most large-scale Java applications. It will likely invalidate *all* mobile apps which use Java at various levels of the standard library.
- Even if useful, assumptions can be difficult to check. In other words, ensuring that the assumptions hold can entail a separate static – or runtime – analysis.
- In the context of practical deployment, it is not entirely clear what to do with these assumptions. Checking these assumptions creates extra static or runtime overhead. When they do not, the user is potentially left without any useful answers. In practice, tool makers will choose to proceed when the assumptions do not hold. While there is value in informing the user that their code violates tool’s assumptions, it is not clear how high that value is. In real-life settings, developers often hardly have the time to fix *errors* that come from a tool, not to mention *warnings.*

Our Proposal

The impact of not handling exception can be especially pronounced in large-scale

In this paper we advocate the following position:

- That "best effort soundness" is a useful concept?
- That unsoundness is unavoidable so we need to characterize levels of it?
- That unsound analyses can be very useful, so soundness is not a goal by itself?
- That there is a well understood concept of "soundy" in every language which should go without much explanation while unsoundness beyond that should be documented?