Martin Kellogg
kelloggm@cs.washington.edu

The first issue I'd like to confront is the difficulty of building good static analyses, since it's what I work on most days. Designing a simple static analysis (like the parity analysis we've been discussing in class) is relatively straightforward. And, we have an excellent mathematical formalism for such an analysis (i.e. abstract interpretation, which we also talked about in class). However, building a more complex static analysis is a time-consuming, manual process; an especially time consuming part is discovering false positives the analysis is producing and then writing sound transfer rules to avoid them. As an example, consider my current research project, which is a type system (static analysis) for preventing out of bounds array accesses. The tool warns at array accesses that might be out-of-bounds, and is implemented as a series of static analyses that each use the results of some or all of the previous ones. The tool is quite complex; I'm not sure I *could* write down a transfer function for the more complex analyses using an abstract interpretation formalism (not that I think it's impossible *per se*, just mind-numbingly difficult and tedious). Despite the lack of formality, I'm actually quite confident in the tool itself, though. And that's because we have an extensive test suite that contains a lot of real-world programs (or program fragments) that we handle correctly. Constructing this test suite wasn't trivial, but it was much easier than building the tool itself. This has led to me to wonder whether there might be a better way of building complex analysis tools like this, and I was reminded of some other interesting research going on at UW in the area of synthesis. James Bornholt and Emina Torlak published a paper in PLDI this year [1] about synthesizing memory models for real architectures (which are very complex) from litmus tests, and I wonder whether something similar could be done to synthesize abstract interpretations for complex analyses from tests. If so, it might even be possible to automatically construct analyses that work on real code by providing examples of the kind of behavior we want to allow and prevent. Doing so would allow developers to deploy custom static analyses for the problems they were facing with relative ease.

Another problem that I encounter regularly is the problem of *flaky tests* [2]. This is a well-known issue in the testing literature: sometimes tests fail not because the code is wrong, but because of some other change in the environment. Sometimes they even fail non-deterministically. All of these problems make testing less efficient, because developers waste time dealing with tests that really ought not to be failing. Further, they encourage developers *not* to test, since the test might end up being flaky sometime later. In small projects, this problem isn't very pronounced, but in large projects (especially with distributed teams, where it might be difficult to talk to the person who wrote the test!) a failing test is a serious problem. Existing solutions to the problem in the literature are unsatisfactory. Many flaky tests are caused by test dependence, and there exist solutions [3] [4] to automatically root out test dependence, but they are expensive and time-consuming to deploy, and even the researchers who develop them don't always use them in practice (as an anecdotal example, [3] is not used by its own authors in all of their other projects, which I know since I work on a project with Mike and we don't use it!). When the tests' flakiness is not caused by test dependency, the best

known technique for dealing with the flakiness is to rerun the test several times, and see if it passes at least once. That's not a very satisfying answer, and, worse, it often needs to be done manually. The case where I most frequently encounter these kind of tests are when running tests that take non-deterministic amounts of time on CI servers; if the test runs quickly, it will pass, but if it takes too long it ends up being killed by the timeout. The timeout could be raised, but doing so is undesirable (since tests that are stalled will take longer to actually timeout and fail!). Other times, flakiness can be caused by concurrency issues or by dependence on the environment. In general, though, flaky tests seem to be caused by tests not being associated with a particular setup in which they should be run. We typically think of a test as just an input and an oracular output, but I would argue that we need some kind of *testing system* (probably implemented as a CI tool) that keeps track of the configuration of the environment explicitly as part of the test. It might even be possible to do so automatically via machine learning - consider that most tests pass when they are first committed (hopefully), and only manifest their flakiness later. If each test was associated with a particular environment, then when flakiness manifests it can be rerun in the environment under which it was known to succeed before.

Another major annoyance that I encounter often is merge conflicts. Good development practice is to develop each change or feature in its own branch, and then merge into master. In theory, this works very well. In practice, though, it is sometimes necessary to build on top of a feature from another branch, and it is always necessary to get code reviewed before it is merged. When these two things co-occur, the result is often messy and time-consuming merge conflicts that really shouldn't exist. Consider this scenario:

1. You develop feature A on branch A, and submit a PR to master.
2. You start working on feature B, which depends on A, on branch B, which starts on A.
3. Another developer reviews A and requests changes; you make changes and A is merged to master.
4. You finish working on B, and want to PR, but you get a merge conflict, because you were working off a different copy of A!

Merging in A after making the changes but before PRing doesn't help; you still get these large merge conflicts. And these merge conflicts are especially frustrating, because they're stupid: different names, different import sets, things like that. So why can't existing tools like git handle these scenarios? The problem, I think, is twofold: git relies on textual diff, first, which makes it difficult to realize when code has moved as opposed to changed; and git doesn't take the shared history of two branches into account when addressing a merge conflict. The first could be solved by a good tree-based diff algorithm. Such algorithms exist [5], but have not been deployed in widely-used version control systems (like git), which instead use diff3 [6]. A tree-based diff algorithm would probably help, but I think most of the problem could be resolved by looking at shared history. If B is based on A, and then changes were made to A and then merged into master, shouldn't we prefer that over the version of A that B started working off of? Even a simple tool that automatically took all commits on B that were different from A and then cherry-picked them onto a new branch automatically would be helpful.

References

[1] Bornholt, James and Torlak, Emina. “Synthesizing memory models from framework sketches and litmus tests.” Programming Language Design and Implementation (PLDI), 2017.

[2] Luo, Qingzhou, et al. "An empirical analysis of flaky tests." Foundations of Software Engineering (FSE), 2014.

[3] Zhang, Sai, et al. "Empirically revisiting the test independence assumption." International Symposium on Software Testing and Analysis (ISSTA), 2014.

[4] Bell, Jonathan, et al. "Efficient dependency detection for safe Java test acceleration." Foundations of Software Engineering (FSE), 2015.

[5] Falleri, Jean-Rémy, et al. "Fine-grained and accurate source code differencing." Automated Software Engineering (ASE), 2014.

[6] Khanna, Sanjeev, Keshav Kunal, and Benjamin C. Pierce. "A formal investigation of diff3." Foundations of Software Technology and Theoretical Computer Science, 2007.