15-745: Optimizing Compilers for Modern Architectures

Project Proposal

Sol Boucher and Goran Žužić

March 18, 2016

+ Interesting idea for a project
+ Good match for the course
+ Good plan & timeline

In trying out by hand, you are likely to find many false positives, and should plan time for coming up with ideas to reduce FP. Also, is there anything to be learned from [4]?

1 Introduction

Logical bugs are a big source of programming errors and can be notoriously hard to catch. While there is no hope to detect all logical mistakes in source code, one might hope to design bug detection software to capture a specific type of probable bugs. To this end, we propose a static bug detection technique that uses dimensional analysis to flag possible programmer mistakes.

Examine the C++ source code in Listing 1. If we remove the statement "/*sq_dist*(...)" marked in red, the code contains a dimensional bug. To see the bug, it is important to know that the variables `v`, `g` and `pts[.]` hold 2D points. Let's denote the dimension of the unit distance of this 2D plane as E. It can be inferred that the dimension of `dot_prod` is E^2. Without the statement in red, the dimension of `alpha` is also E^2. But this implies that `pts[i].x + alpha*v.x` will be adding together variables of dimension E and E^3, a clear dimensional bug. The proposed method vows to catch such bugs.

Listing 1: Example of a dimensional bug

```
double dot_prod = (g.x-pts[i].x)*v.x + (g.y-pts[i].y)*v.y;
double alpha = dot_prod / sq_dist(v, {0, 0});
if (...) {
  Pt cand = {pts[i].x + alpha*v.x, pts[i].y + alpha*v.y};
  ...
}
```

This example from Listing 1 comes from a competitive-programming event (edited for clarity) and this was the actual bug one of the authors made during the contest.

2 Scope and Variations

We propose a static bug detection method that uses dimensional analysis. The method is purely non-parametric and doesn't require any additional annotation from the programmer. While such dimensional bugs can often be found easily with unit testing, there are certain scenarios where we believe our method can be useful: *i)* in cases where unit-tests don't cover

This argument is a bit weak... can you find other cases where unit testing is not the answer?

all code paths or where the unit-test use trivial values (0 or 1) for dimensional variables *ii)* in the context of competitive programming for fast static bug detection.

The approach is static (compiler-assisted) and is useful for debugging. Therefore we believe it is well within the scope of the 15-745 course.

Before we describe the methodology of the method itself, it is worth mentioning that such a dimensional analysis is a fairly novel approach. The closest prior art we were able to find are articles describing software packages for dimensional annotations [4] or semantic debugging utilizing various approaches [1, 3, 2].

Below, we provide our goals for this project. If the problem is more challenging than we expect, we will only complete the 75% goals; if the project is roughly as challenging as we expect, then we will complete some subset of the 100% goals; and, if we make progress at a quicker rate than anticipated, we will consider some of the 125% goals.

75% goal If the dimensional analysis proves too hard or produces too many false-positives to be useful we will consider a reduced-scope problem of "mod-consistency-detection". Basically, each time a variable stores a value ~~that~~ modulo some P, we are to expect that this variable will always contain some value calculated modulo P. If it appears this is not the case, we issue a warning. This problem can be viewed not only as a semantic bug detection, but also as a dimensional analysis problem if we consider "int modulo P" as a separate type.

100% goal We would like to have a working dimensional analysis tool that will produce useful debugging output when given as input a contest problem with a dimensional analysis bug. Additionally, if the framework appears too-impractical, we might consider adding some programmer-annotation tools to help guide the analysis.

125% goal As our most ambitious goal, we will try to design a practical framework that could be fed mature software codebases and have a reasonable output.

3 Resources and Project Timeline

We plan to create a proof-of-concept implementation in the form of one or more analysis passes for the LLVM compiler framework; specifically, we intend to operate on LLVM IR in the hopes of keeping the code applicable to multiple languages. That said, for the purposes of this course, we only intend to test either C or C++, leaving support for other languages as future work.

Because the LLVM bytecode uses SSA form, each variable in the high-level language maps to a group of registers. Our analysis requires treating such a register pool as a single logical storage area; for this reason, we'll need access to some information from the compiler frontend. Fortunately, such register associations are part of the information a debugger needs when stepping through code at runtime, so we intend to make use of debugging annotations created by the frontend. This debugging information will also be useful for another reason: once we've detected a possible semantic bug, we'll want to report information—probably including the variable name and relevant line numbers—to the user.

As a competitive programmer, Goran has an extensive dataset of programs with a wide range of bugs, several of which exhibit the kinds of errors we're interested in. At the beginning of the project, he can be collecting real code samples and generating trivial tests to produce datasets to aid with development and use for evaluation. Simultaneously, Sol will be able to investigate the LLVM debugging annotations and begin using them to associate related registers.

Week of	Task
3/21	Become familiar with LLVM IR debugging information, Finish collecting basic test/evaluation dataset and try the algorithm out by hand
3/28	Be able to group registers by variable and represent variables' dimensionalities, Enumerate rules for updating dimensionalities based on the arithmetic operations present
4/4	Be able to partially analyze toy examples and detect obvious dimensionality mismatches Start to get a sense of the false negative/false positive rate
4/11	**Milestone report due.**, Continue development, Develop basic heuristics to prune false positives if needed
4/18	Achieve enough functionality to be able to run on arbitrary code samples, Work on presenting suspected errors reasonably to the user
4/25	**Poster session at end of week.** Experiment on our full dataset, Prepare poster, and Perform a case study in actually debugging unfamiliar code with the tool
5/2	**Final report due.** Produce final report

4 Experiments and Evaluation

As an evaluation method we are considering downloading a vast amount of source codes from open competitive programming sites like `codeforces.com` and finding dimensional bugs. Obvisouly, such evaluation will require a lot of manual labor of checking for false-positives and it is not clear what is the probability that a random source code contains a dimensional-analysis bug. However, our hope is to find a nontrivial number of bugs and present a case study in which we use the tool to find them.

5 Outline of the Paper

The final paper will proceed roughly as follows:

- Abstract
- Related work
  - Semantic types
  - Dimensionality
- Problem
- Approach
- Evaluation

  - Sample set
  - Incidence of semantic type bugs
  - Incidence of dimensionality bugs
  - Blind debugging
- Results
  - False positive rate
  - False negative rate
  - Performance cost
  - Blind debugging successfulness
- Conclusion
- Future work

References

[1] F. Bourdoncle. Abstract debugging of higher-order imperative languages. *Proceedings of the ACM SIGPLAN 1993 conference on programming language design and implementation*, 28:46–55, 1993.

[2] J. Choi. Parallel program debugging with flowback analysis. Technical report, Madison, WI (US); Univ. of Wisconsin, 1989.

[3] C. Hall, K. Hammond, and J. ODonnell. An algorithmic and semantic approach to debugging. In *Functional Programming, Glasgow 1990*, pages 44–53. Springer, 1991.

[4] P. N. Hilfinger. An ada package for dimensional analysis. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 10(2):189–203, 1988.