PROGRAM ANALYSIS: AN EXPLORATION OF RELATIONAL VERIFICATION

A Thesis
Presented to
The Academic Faculty

by

Nicholas K. Ryan

In Partial Fulfillment
of the Requirements for the Degree
Bachelor of Science with the Research Option in the
School of Computer Science

Georgia Institute of Technology
May 2017

PROGRAM ANALYSIS:
AN EXPLORATION OF RELATIONAL VERIFICATION

Approved by:

Professor William Harris, Advisor
School of Computer Science
Georgia Institute of Technology

Professor Mayur Naik
School of Computer Science
Georgia Institute of Technology

Date Approved: May 3, 2016

TABLE OF CONTENTS

PREFACE

This thesis is submitted in partial fulfilment of the requirements for a Bachelor's Degree in Computer Science. It contains work done from January to May 2016. My supervisor on the project has been Dr. William R. Harris, School of Computer Science, Georgia Institute of Technology. This thesis has been made solely by the author; the algorithms for which these benchmarks were collected, however, were designed by others.

ACKNOWLEDGEMENTS

I would like to thank my supervisor Dr. William R. Harris for his guidance with this work and providing insightful feedback as necessary.

Thanks to Dr. Mayur Naik for his assistance and feedback on this thesis.

And finally, thanks to Dr. Malavika Shetty for her unending support and feedback from the beginning of the proposal to this thesis.

Program Analysis:
An Exploration of Relational Verification

Nicholas K. Ryan

8 Pages

Directed by Professor William Harris

Program analysis is a quickly growing field. We attempt to tackle new problems regarding program equivalence, concurrent programs, and information flow. As these problems have not been addressed at the same level we desire, there are no adequate testing suites. Previous tests for program analysis generally consisted of 'toy programs' which was not a trend we wanted to follow. As such we began exploring new resources for testing such as coding practice problems, industry code, and related academic research. We found many unique testing materials which satisfy many program analysis problems as the code we analyzed was 'real-world' code. By 'real-world' we mean code which was not written purely for testing. This code was written to solve a problem in industry or research with no knowledge of our analysis. In the end we hope to provide a database of benchmarks which can be used for future projects by the program analysis community.

CHAPTER I

INTRODUCTION

Program analysis is a broad field in which the behavior of the program is analyzed through an automated process. In general, these programs are examined for certain properties such as correctness, robustness, safety, and liveness. Program analysis can typically be considered as one of two categories: program optimization or program correctness; however, these categories are not strictly mutually exclusive. Program optimization focuses on decreasing the amount of resources used and improving the speed at which the program can operate. Program correctness is concerned with determining if a program accomplishes its desired goal. We choose to initially focus our benchmark collection on three subfields: program equivalence, analysis of concurrent programs, and information flow.

1.1 Program Equivalence

Previous work in program analysis has focused on relatively simple programs, generally it involves a single program. We break this trend by examining relational verification, specifically program equivalence which requires analyzing two programs simultaneously. Although some work has been done in this subfield, it has all been rudimentary. It could only show equivalence when two programs are of similar structure [1], single-threaded, or analyzed during compilation [3]. We introduced our own algorithm which could handle varying structures and which does not rely on the compiler.

Program equivalence is an undecidable problem; this means that it is impossible to confirm equivalence for all inputs so there are some program combinations that we cannot say "Yes, they are equivalent" or "No, they are not equivalent." This is due in

part to an infinite number of inputs to these programs, making it infeasible to test all of their inputs. Because it is undecidable, any work in this field is ground breaking. However, because of the difficulties that come with this problem it can be difficult to adaquately test any new algorithms. Many testers resort to 'toy programs' which only test the desired property. We, however, intend to test programs written without knowledge of these tests. We intend to test 'real' programs. Our collection method is described below in Procedures.

1.2 Concurrency

As mentioned above, program analysis is still in relatively simple stages. There has been little work regarding concurrent programs so we hope to make substantial steps in this area. As such initial steps will not be large and we choose to focus on various data structures and hope to prove various properties such as dead-locks and race conditions. Continuing our trend of avoiding 'toy programs' we collected our concurrent programs from industry and academia projects. If possible we would like these benchmarks to also satisfy the equivalence requirements. So if we could find multiple implementations of the same type of structure that would be ideal. This desire led us to the Synchrobench study [2]. This work provided a large supply of concurrent data structures, written for testing, that satisfied equivalence requirements as well.

1.3 Information Flow

This area is still extremely new for program analysis. We hope to apply Information Flow techniques to program analysis as a new way of approaching certain properties of the program. This can open new doors for analysis by providing a different look at the security features of a program while still proving properties relevant to concurrency or equivalence. However, we have not made significant progress in this field so we hope that finding benchmarks will provide insight into approaching the problem.

CHAPTER II

PROCEDURES

2.1 Finding Relational Verification Benchmarks

In general, we find all of the benchmarks in a similar manner. We collect them through coding practice websites, popular git repository websites, and well-known benchmark repositories. We collect them from code that is already established and tested. We would rather make use of fully developed code than write our own 'toy' programs to prove the feasibility of our algorithms. Each relational verification problem will have its own unique traits and properties that we would like to analyze. As such we may need to search for these benchmarks in different ways.

The only requirement which holds across all benchmarks is that they **must** compile to the JVM. We have this requirement as some of the most powerful analysis tools apply only to JVM bytecode. Furthermore, the JVM allows a consistency across architectures that cannot be obtained with other languages.

2.1.1 Program Equivalence

In order to develop sufficient tests we need to consider the problem were trying to solve and its limitations. Program equivalence is not a trivial task. As these are just the beginning steps to solving an undecidable problem, we ignore most non-trivial cases. In order for a program to be a good test case we need to satisfy a few requirements:

- The programs need to be 'primitive.' The addition of linked data structures such as ArrayLists or HashSets attempts to tackle a more complex problem which we need not consider at this point in our research. Therefore programs with linked data structures are not considered appropriate tests.

- The code should vary in complexity. Requiring files with branching, loops, or arrays, but also basic straight-forward files as control.

- The problems must be algorithmic in nature to avoid ambiguous solutions. There must be multiple approaches to the solution.

In consideration of these requirements we find that practice interview problems fit quite well. These problems can be found on popular practice websites such as CodeChef and TopCoder. The problems are simple enough to have an elegant solution however there are multiple ways to approach the problem. In addition, it's helpful that each solution can generally be contained to a single file. Next steps include accessing the Software-artifact Infrastructure Repository (SIR) for more complex, well-known, program analysis benchmarks. SIR will allow testing across multiple versions of 'larger, more advanced' programs to test equivalence when the only change should be optimization. The inclusion of these benchmarks will further validate our findings as it's well-known that benchmarks from this repository hold up under extreme scrutiny.

It was found that searching GitHub did not result in strong benchmarks.

2.1.2 Concurrency

The only requirement for concurrent benchmarks was that they must be data structures. This allows us to focus on testing the data structure instead of being concerned if it solved a certain problem. For testing we needed to verify concurrent properties for data structures, such as dead lock and data races, in addition to all of the properties which should hold for a 'normal' data structure, such as null pointer and array index out of bounds exceptions.

We found that the best source of benchmarks came from GitHub. Searching for concurrent libraries resulted in plenty of large repositories containing concurrent data structures.

2.1.3 Information Flow

Although we are still searching for benchmarks, we believe the best place to find these benchmarks will be GitHub. As of now, we do not know what will make a good benchmark or what we're trying to find. Our goal is to find programs which leak 'secret' information to the 'public.' This can be any information released to a common out-source that provides insight into the information not released. This means we're concerned with programs that will handle sensitive information such as healthcare or banking applications.

CHAPTER III

RESULTS AND DISCUSSION

The most prevalent benchmarks were dependent on which problem we we're trying to solve. We provide a breakdown of each problem and what we have found or hope to find.

3.1 Program Equivalence

The benchmarks for program equivalence primarily came from the practice programming websites. We pulled approximately 850 problems and 95000 solutions to these problems. This will give us plenty of cases to test our algorithm. At this point we found that our algorithm, while efficient, does not prove strong properties. As such, we have not progressed farther than these practice problems in our testing.

3.2 Concurrent Programs

Our primary focus for concurrency was in datastructures. This allows us to focus on testing the data structure and less on whether the code correctly solves some arbitrary problem. These repositories came from large scale industry programs, research papers, and hobbyists. This gives good opportunity to see how different code sources may be designed to meet different specifications. A large proportion of our benchmarks for concurrency came from the Synchrobench repository[1]; which consisted only of concurrent datastructures. Each datastructure is written in multiple concurrent paradigms and as such will be useful for testing equivalence down the road. At this

[1]https://sites.google.com/site/synchrobench/

point in time we have also drawn two maps from JCommon[2], a collection of Facebook code, a FastArrayBlockingQueue from a repository called Concurrent[3], and four structures from MapDB[4].

3.3 Information Flow

Information flow benchmarks are still being found. However, we believe that once these benchmarks are found it can open doors to ideas for creating algorithms relevant to program analysis.

3.4 Database

Our goal for this work was to create a new benchmark database for the program analysis community. We hope that this database will serve as a valuable tool for future work. Furthermore, we hope that a highly-accesible benchmark database will give rise to stronger algorithms as they can be more rigorously tested.

[2]https://github.com/facebook/jcommon
[3]https://github.com/coderplay/concurrent
[4]https://github.com/jankotek/mapdb

REFERENCES

[1] B. Godlin, O. S., "Regression verification : Proving the equivalence of similar programs," *Software Testing, Verification, and Reliability*, vol. 23, pp. 241–258, 2013.

[2] Gramoli, V., "More than you ever wanted to know about synchronization: Synchrobench, measuring the impact of the synchronization on concurrent algorithms," in *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'15)*, pp. 1–10, ACM, Feb 2015.

[3] Necula, G. C., "Translation validation for an optimizing compiler," *ACM SIGPLAN Notices*, vol. 35, no. 5, pp. 83–94, 2000.