Detecting Runtime Check Patterns and Applying Optimization in GVC0

Sam Estep
Carnegie Mellon University
Pittsburgh, PA, USA
estep@cmu.edu

Paulo Canelas
Carnegie Mellon University
Pittsburgh, PA, USA
pacsantos@cmu.edu

1 PROJECT WEB PAGE:

https://pcanelas.com/optimizing-compilers

2 PROJECT DESCRIPTION

Gradual verification [2] is a technique that allows developers to incrementally and progressively verify their system by introducing partial specifications. Specifications in gradual verification are statically and dynamically verified. Statically, the verifier assumes any strenghtening of imprecise specifications, and to guarantee soundness, it generates dynamic checks when the partial specifications are optimistically strengthened.

GVC0 [3] is an existing technique for gradual verification is gradual verification which allows one to introduce partial specifications. These partial specifications contain static verifications and checks that can only be checked at runtime that are inserted into GVC0. GVC0 is a tool that works on C0 [1], a C version used for education, augmented with specifications for gradual verification. In our work, the idea is to take these GVC0 programs and optimize the runtime checks performed.

To this end, this work intends to answer the following two research questions.

RQ1. What are the runtime check patterns introduced by GVC0?

RQ2. To what extent do optimizations to runtime checks impact program performance?

To answer RQ1, we intend to execute GVC0 on the datasets presented in prior work [3, 4], manually inspect the generated code with the injected execution checks, study the patterns in which these are introduced, and figure out how they can be optimized. Regarding RQ2, we plan to use the detected patterns to improve the generated code. We will perform the optimization by either (a) changing the way runtime checks are introduced by GVC0, or (b) developing LLVM passes that detect a pattern and perform an optimization. Finally, we will evaluate the impact of the optimizations by comparing the runtime performance of the original code with and without optimizations.

We define the goals of the project as follows.

75% Goal: Detect and describe the patterns present in GVC0's output. Implement one optimization and compare the performance between the optimized and unoptimized code.

100% Goal: Same as the 75% goal, plus an extra optimization. Perform an ablation study, and check if performance changes as optimizations are applied.

125% Goal: Same as the 100% goal, plus implement remaining optimizations depending on the number of patterns discovered.

3 LOGISTICS

3.1 Plan of Attack and Schedule

Week 1: Download and build the GVC0 tool. To allow replication of the results, create a virtual machine or Dockerfile that automates the setup. Execute the GVC0 tool on its dataset and obtain the outputs to be analyzed. Develop extra test cases, in case the dataset does not contain enough use cases to perform the qualitative study.

Week 2: Manually inspect each compiled program, analyze their patterns and aggregrate use cases according to the patterns encountered. This first step answers RQ1, and provides a dataset of labeled programs with their patterns.

Week 3: Select one of the patterns and implement the optimization by changing GVC0 or creating an LLVM pass.

Week 4: Perform the evaluation and compare it against the GVC0 generated code without optimizations. Write the milestone report.

Week 5: Select another pattern and implement the optimization. Perform an ablation study to check the impact when introducing the optimizations.

Week 6: If possible, implement any remaining possible optimizations and evaluate them (125% Goal). Write the final report and the poster.

3.2 Milestone

By the milestone, we expect to have executed GVC0 over a dataset and obtained a set of programs, qualitatively analyzed these programs and detected a set of patterns that can be exploited to perform optimizations. Futhermore, we intend to have developed an optimization according to one of these patterns, and quantitatively evaluated the performance of the program with and without the optimization.

3.3 Literature Review & Resources

To conduct this work, we studied prior work related to gradual verification [2, 4], and the work related to GVC0 [3]. We also contacted the authors of GVC0, to ensure that there are possible optimizations that can be introduced in the tool. Furthermore, we got access the GVC0 tool.[1]

[1] https://github.com/gradual-verification/gvc0

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

3.4 Getting Started

We have recently started trying to setup GVC0.

REFERENCES

[1] Rob Arnold. 2010. *C0, an imperative programming language for novice computer scientists.* Ph. D. Dissertation. Master's thesis, Department of Computer Science, Carnegie Mellon University.

[2] Johannes Bader, Jonathan Aldrich, and Éric Tanter. 2018. Gradual program verification. In *Verification, Model Checking, and Abstract Interpretation: 19th International Conference*. Springer, 25–46.

[3] Jenna DiVincenzo, Ian McCormack, Hemant Gouni, Jacob Gorenburg, Mona Zhang, Conrad Zimmerman, Joshua Sunshine, Éric Tanter, and Jonathan Aldrich. 2022. Gradual C0: Symbolic Execution for Efficient Gradual Verification. arXiv:2210.02428 [cs.LO]

[4] Jenna Wise, Johannes Bader, Cameron Wong, Jonathan Aldrich, Éric Tanter, and Joshua Sunshine. 2020. Gradual verification of recursive heap data structures. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–28.