Optimization of a Symbolic-Execution-based Program Verifier

Bachelor's Thesis Description

Moritz Knüsel
Supervised by Prof. Peter Müller and Dr. Malte Schwerhoff

April 12, 2019

Introduction

Viper is a language and a set of tools developed at ETH, providing an architecture on which new verification tools and prototypes can be developed simply and quickly. Viper uses permissions to reason about programs with mutable state. Viper provides two backends for verification, one based on verification condition generation, and another based on symbolic execution.

Users of Viper have noticed that some of the inputs they generate can take an unusually long time to verify. Often it is unclear why these inputs take longer to verify than others. The goal of this project is to answer this question for Silicon, the symbolic-execution-based backend of Viper.

First of all, examples of problematic Viper code will need to be collected. These will mainly be sourced from existing frontends. These examples will then be categorized and the workings of Silicon on these inputs will be analyzed. Based on that, improvements to Silicons performance will be attempted.

Core Goals

1. Collect Examples

The first goal is to collect examples of problematic inputs. These will mainly be sourced from:

- **Users and Developers of Viper**

  Frontend developers will be asked about examples of problematic inputs they have encountered in the past. They may also be able to point out potential metrics to consider in the categorization, based on their experiences.

- **Problematic Inputs for Carbon**

  Philippe Voinov is currently working on optimizing Carbon[1], and may be able to share his collected inputs to see if they pose problems to Silicon as well.

- **Constructed Examples**

  If the categorization implies certain constructs to be problematic, artificial inputs may be written to better understand what it is about these constructs that slows down verification.

Having such a collection will not only be useful to identify potential sources of performance loss in Silicon, but also to evaluate if a modification is actually beneficial, by benchmarking against the collection.

2. Categorize Examples

Once a set of problematic inputs has been assembled, various metrics will be computed in an effort to reveal patterns in the inputs and gain insights into Silicons performance. Such metrics may include:

- The amount of Viper code compared to the generated SMT code
- The number of branches taken during the symbolic execution
- The time spent in Z3 compared to the time spent in Silicon
- The number of state consolidations
- The occurences of specific features of Viper

Furthermore, problematic inputs will be compared to faster inputs of similar size, to see if and where their characteristics differ.

3. Improve Silicon

Based on the insights gained by looking at the problematic inputs, further research into potential optimizations will be made. This will be an iterative process of identifying problems, changing Silicon and evaluating the impact of the changes. The different categories of inputs will be ranked, based mainly on how bad they perform and how promising they are with respect to improvements. This ranking determines what will be investigated first. In order to evaluate the changes, Silicon will be benchmarked on the collected inputs and its regular test suite. Potential changes include the following:

- Silicon's core
  - Heap encoding

[1] www.ethz.ch/content/dam/ethz/special-interest/infk/chair-program-method/pm/documents/Education/Theses/Philippe_Voinov_BA_description.pdf

  - State consolidation
  - Branching and joining paths
- SMT
  - Resource bounds instead of timeouts
  - Query caching
  - Different solvers
- Architecture
  - Z3 Java bindings instead of stdio
  - More appropriate collections and data structures

Extension Goals

1. Utilize more Parallelism

Another potential performance improvement would be to utilize more parallelism, for instance on branching during the symbolic execution. This would most likely require using the Z3 Java bindings.

2. Heuristics for Backend Selection

If it turns out that there are certain classes of inputs where Carbon is very fast while Silicon is very slow, or vice-versa, it may be beneficial to come up with heuristics to try and choose the faster backend automatically.

3. Work on Less Promising Issues

If some of the issues identified in the last core goal don't look very promising, they can still be investigated if there is time.