Optimization of a Viper-based verifier

Bachelor Thesis Project Description

Till Arnold
Supervised by Prof. Dr. Peter Müller, Vytautas Astrauskas
Department of Computer Science
ETH Zürich
Zürich, Switzerland

I. Introduction

Rust is a system programming language that statically guarantees memory safety through its type system. This type system eliminates otherwise common classes of bugs and security vulnerabilities such as dangling pointers, data races and buffer overruns. However Rust's type system cannot ensure the general functional correctness of programs. Such correctness can be checked by static verifiers. Traditional static verifiers for system programming languages require an expert to provide lengthy and complicated program specifications to allow for the formal verification of programs. In contrast, Prusti [1] — a verifier for Rust — aims to require much simpler specifications that can be provided by the programmers. To accomplish this Prusti leverages Rust's type system for translating the Rust code into an intermediate verification language called Viper [2]. The Viper suite of tools is then used to verify the functional correctness of the original Rust program. An example of a Rust program annotated with Prusti attributes can be seen in Listing 1.

As Prusti is targeted at Rust programmers, with the goal of allowing formal verification of Rust programs while being as unobtrusive as possible, the quality of the Prusti tool is of utmost importance. In particular the performance of Prusti is crucial because slow verification times discourage the use of the tool.

One of the main contributing factors to the verification time in Prusti is the quality of the Viper code generated by Prusti. Verification times of under 10 seconds would be desirable. It is to be assumed that semantically equivalent but more efficient code could be generated. Table I shows some simple measurements of the average verification time of the generated Viper code. These measurements were obtained as described in section V Testing methodology.

This thesis aims to investigate the aforementioned process of translating Rust to Viper and to identify, analyze and mitigate sources of slowdown in the generated Viper code.

As a starting point, we plan to look into *two categories of potential performance improvements*:

1) Removal of unnecessary elements of a generated Viper program:
   - Removing unused predicates and bodies of predicates that are never unfolded.
   - Removing dead variables and trivial statements such as `inhale true` or `assert true`.
   - Merging and reordering control-flow graph nodes. While the positive performance impact of this is probably negligible it makes the generated Viper code more human-readable and potentially facilitates the discovery of more optimization potential.
2) Purification: Reducing the number of heap allocated variables. Since Rust allows calling functions with references to stack variables, Prusti models nearly all variables as allocated on the Viper heap. If no reference to a certain variable is ever created that variable could be modeled without a heap allocation. It is to be assumed that Viper code that uses regular Viper variables that are pure instead of heap-allocated ones would have better performance.

TABLE I
Verification Runtime of Selected Rust Examples

Filename	Avg. runtime in seconds
`account.rs`	1.581
`Binary_search.rs`	7.738
`fibonacci.rs`	2.772
`knapsack.rs`	97.600
`Knuth_shuffle.rs`	2.816
`Selection_sort.rs`	16.313

II. Core Goals

The goal of this thesis is to identify which patterns of the generated Viper code cause slow verification times, analyze why these specific patterns are problematic, and improve the Prusti implementation so that these patterns are avoided.

1) **Create a test suite that demonstrates the performance issues:** To investigate the performance characteristics of the generated Viper code a sufficiently large body of test cases is needed. These test cases form the basis of all further analysis.

```
struct Account { bal: u32 }

impl Account {
  #[pure]
  fn balance(&self) -> u32 { self.bal }

  #[ensures(self.balance() ==
  old(self.balance()) + amount)]
  fn deposit(&mut self, amount: u32) {
    self.bal = self.bal + amount;
  }

  #[requires(amount <= self.balance())]
  #[ensures(self.balance() ==
    old(self.balance())  - amount)]
  fn withdraw(&mut self, amount: u32) {
    self.bal = self.bal - amount;
  }

  #[requires(amount <= self.balance())]
  #[ensures(self.balance() ==
    old(self.balance())  - amount)]
  fn transfer(&mut self, other: &mut Account,
  amount: u32) {
    self.withdraw(amount);
    other.deposit(amount);
  }
}
```

Listing 1: Simple example for the attributes Prusti uses to annotate functions. This example is taken from `account.rs` from the Prusti tests

2) **Analyze the test cases in the test suite and identify potential sources of poor performance. Utilizing the insights gained from the analysis, add test cases to the test suite to strengthen the understanding of the underlying performance issues:** Identify patterns in the generated Viper code that could be causing the slow verification. For each of the identified patterns, including the ones in the *two categories of potential performance improvements*, propose reasons as to how this slows down the Viper verifier. Steps 1) and 2) might have to be iterated multiple times to assure sufficient insights into the source of the underlying performance issues.
3) **From the previously identified sources of poor performance, select the most important and promising ones and implement mitigations for those:** This might be accomplished by improving the generation of Viper code directly or by introducing an optimization pass on the Prusti intermediate representation (VIR). This implementation should aim to mitigate the performance issues by applying the knowledge gained in the previous steps.
4) **Test the implementation:** Run the test suite on the improved implementation, compare these results to the results of the original implementation and analyze why and by how much the performance increased.

III. Extension Goals

1) **Setup CI for testing performance regression:** To avoid future changes to the code base (re)introducing performance problems without this being a conscious decision, a step in the CI could be introduced to keep track of the performance over time.
2) **Implement other performance optimizations:** Aside from the quality of the generated Viper code there are other aspects of the Prusti code base where performance can be improved. For example, one could profile Prusti to identify where it spends the most time while generating the Viper encoding and then improve the corresponding implementation or algorithms.
3) **Analyze performance impact on entire crates:** This allows measuring the improvements of the implemented optimizations on real world code bases.
4) **Implement a Clippy like tool to point out performance sensitive patterns:** Certain patterns of Rust code may identifiably cause performance issues in Prusti, with the implemented optimizations providing little or no performance improvement. For these cases a tool could be written which points out to the programmer problematic sections of code that will significantly slow down Prusti verification.
5) **Reason about soundness of optimizations:** Variably formal reasoning about the correctness of the implemented optimizations could be incorporated into the final thesis.
6) **Investigate advanced optimization techniques:** It might be possible to split a Viper program into multiple smaller Viper programs that are faster to verify than the original Viper program. It would be particularly interesting to find a way to split a Viper program into (nearly) disjunct parts such that there is little to no overlap. These smaller Programs would potentially also benefit more from the speedup gained from other optimizations.
7) **Reduce the number of fold/unfold statements by inhaling the permissions in the expected shape:** Some fold/unfold statements could be eliminated by, for example, inhaling the body of the predicate directly instead of inhaling a predicate and then unfolding it.

IV. Schedule

- Goals 1) and 2) 3 weeks for test creation and analysis
- Goal 3) 8 weeks for implementation
- Goal 4) 2 weeks for evaluating results

- Extension Goals 3 weeks
- Writing report 5 weeks

V. Testing methodology

The average runtimes in Table I where calculate on an Intel i7-4790K CPU @ 4.00GHz with 24GiB DDR3 @ 1600 MHz running Fedora 32 (Kernel `5.8.14-200.fc32.x86_64`). Running ViperServer at commit `8fea4ab61f2793dfbfbdc8ba82cc53df38363271` on OpenJDK 64-Bit `1.8.0_265`, `viper_client` at commit `728c039bfa67f5d438c4d0d9232738864d89ba5a` and Prusti at commit `a147c2fced8bf4b1b7e38fb5eb00e1827cff667e`. The reported runtimes are the average of 18 tests ran with `viper_client` against ViperServer where each time the 3 first test where discarded for warm up. The Viper code was generated from the corresponding Rust files using the above mentioned commit of Prusti. Following is a list of links to the Rust files:

- https://github.com/viperproject/prusti-dev/blob/a147c2fced8bf4b1b7e38fb5eb00e1827cff667e/prusti-tests/tests/verify/pass/demos/account.rs
- https://github.com/viperproject/prusti-dev/blob/a147c2fced8bf4b1b7e38fb5eb00e1827cff667e/prusti-tests/tests/verify_overflow/pass/rosetta/Binary_search.rs
- https://github.com/viperproject/prusti-dev/blob/a147c2fced8bf4b1b7e38fb5eb00e1827cff667e/prusti-tests/tests/verify/pass/quick/fibonacci.rs
- https://github.com/viperproject/prusti-dev/blob/a147c2fced8bf4b1b7e38fb5eb00e1827cff667e/prusti-tests/tests/verify/pass/quick/knapsack.rs
- https://github.com/viperproject/prusti-dev/blob/a147c2fced8bf4b1b7e38fb5eb00e1827cff667e/prusti-tests/tests/verify/pass/rosetta/Knuth_shuffle.rs
- https://github.com/viperproject/prusti-dev/blob/a147c2fced8bf4b1b7e38fb5eb00e1827cff667e/prusti-tests/tests/verify_overflow/pass/rosetta/Selection_sort.rs

References

[1] V. Astrauskas, P. Müller, F. Poli, and A. J. Summers, "Leveraging Rust types for modular specification and verification," in *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, vol. 3, no. OOPSLA. ACM, 2019, pp. 147:1–147:30.

[2] P. Müller, M. Schwerhoff, and A. J. Summers, "Viper: A verification infrastructure for permission-based reasoning," in *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, ser. LNCS, B. Jobstmann and K. R. M. Leino, Eds., vol. 9583. Springer-Verlag, 2016, pp. 41–62.