Translating Pedagogical Verification Exercises to a Rust Verifier

Bachelor Thesis Project Description

Patrick Muntwiler
Supervised by Prof. Dr. Peter Müller, Aurel Bílý
Departement of Computer Science
ETH Zürich
Zürich, Switzerland

October 2022

1 Introduction

Deductive program verifiers are tools that take as input a program and its specification, which must be written formally and unambiguously, and decide whether the program satisfies the specification. At the Programming Methodology Group in ETH Zurich, the Viper verification language [1] was developed, a simple sequential, object-based, imperative programming language that allows the users to specify the behaviour of methods with logical assertions. Given an annotated Viper program, the Viper tooling determines automatically whether the code satisfies its specification or not.
Viper was designed to be an intermediate verification language. In fact, there are multiple front-ends of Viper for modern programming languages, including Gobra [2] for Go, and Prusti [3] for Rust [4]. These tools allow the user to annotate programs written in mainstream programming languages and to verify them by translating the source-code and the specification to Viper. If the obtained Viper file is successfully verified, then the original program is guaranteed to satisfy its specification; otherwise, an error message indicating the source of the verification error is presented at the level of the original source code.
Program verifiers are extremely effective at proving the absence of bugs. Nonetheless, program verification is still regarded as an obscure and challenging field that suffers from a steep learning curve and a lack of learning resources aimed at complete beginners. The focus of this project is thus to develop new pedagogical material for Viper and its front-ends, in order to lower their entry barrier for beginners in verification. To that end, the plan is to write new examples for Viper and its front-ends (this thesis project focuses on Prusti) that exercise features commonly found in deductive verifiers such

as formal specification, mathematical data types, ghost code, and techniques to reason about heap-allocated data structures. Using these features, the goal is to demonstrate how to prove different kinds of properties of programs such as memory safety, functional correctness, and termination. As a starting point, the plan is to translate a subset of the examples and exercises taken from a recent book on Program Verification called "Program Proofs" by K. Rustan M. Leino. The book uses Dafny [5], a programming language and verifier originally developed at Microsoft Research.

2 Approach

The main goal of this thesis is to provide helpful examples for beginners to learn program verification. Another benefit of this thesis is identifying missing features in Prusti that make proofs cumbersome or impossible compared to Dafny.
The plan is to go through the book and to translate examples and exercises from Dafny into Rust code with Prusti annotations.
Figure 1 shows an example of a Dafny function and Figure 2 show the code translated into Rust:

```
method sum(n: int) returns (res: int)
 requires 0 <= n;
 ensures  res == n * (n + 1) / 2;
{
    res := 0;
    var i: int := 0;
    while(i <= n)
        invariant i <= (n + 1);
        invariant res == (i - 1) * i / 2;
        decreases n - i;
    {
        res := res + i;
        i := i + 1;
    }
}
```

Figure 1: A Dafny method that calculates the sum of the first n integers. Dafny checks if the code satisfies the postcondition given the precondition, i.e. the result is $n*(n+1)/2$ if $n \geq 0$.

```rust
#[requires(0 <= n)]
#[ensures(result == n * (n + 1) / 2)]
pub fn sum(n: i64) -> i64 {
    let mut res: i64 = 0;
    let mut i: i64 = 0;
    while i <= n {
        body_invariant!(res == (i - 1) * i / 2);
        res = res + i;
        i += 1;
    }
    return res;
}
```

Figure 2: The previous Dafny function translated to Prusti. Note that overflow checks were disabled for the Prusti code, since Dafny works with unbounded integers and Prusti uses bounded integers.

The examples from the book are then categorized based on if and how their translation worked. Some possible categories are:

- Translation works mostly 1-to-1
- Translation works, but is written differently in Prusti compared to Dafny
  - Example was easier/harder to encode in Prusti
- Translation into Prusti is cumbersome or impossible
  - Subcategories based on the encountered difficulty or missing Prusti feature

One interesting aspect will be the effects of Rusts ownership model on the translations, which may decrease the specification overhead for static analysis in Rust by having memory safety annotations included in the language itself.

2.1 Core Goals

- Translating a subset of Dafny examples and exercises from the aforementioned book into Prusti.
- Categorizing the examples based on their Prusti translation and listing problems found during translation.
- Using the translated examples to improve tutorial-level documentation for Prusti.

2.2 Extension Goals

- A potential extension goal is to implement one of the missing features identified during the project, e.g., enabling Prusti to do termination checks by forwarding the corresponding functionality from Viper.
- Another possibility would be to write a Prusti evaluation, e.g. a verified hashtable.
- Also possible would be to forward the unbounded integer functionality from Viper for use in Prusti ("Ghost integer").

3 Approximate Working Schedule

Task Description	Time
Reading book, identifying examples, translation, categorization	8 Weeks
Evaluation	2 Weeks
Extension Goals	4 Weeks
Write final report	4 Weeks

References

[1] Peter Müller, Malte Schwerhoff, and Alexander J. Summers. "Viper: A verification infrastructure for permission-based reasoning". In: *International conference on verification, model checking, and abstract interpretation*. Springer. 2016, pp. 41–62.

[2] Felix A. Wolf et al. "Gobra: Modular specification and verification of go programs". In: *International Conference on Computer Aided Verification*. Springer. 2021, pp. 367–379.

[3] Vytautas Astrauskas et al. "Leveraging Rust types for modular specification and verification". In: *Proceedings of the ACM on Programming Languages* 3.OOPSLA (2019), pp. 1–30.

[4] Nicholas D. Matsakis and Felix S. Klock. "The Rust Language". In: *Ada Lett.* 34.3 (Oct. 2014), pp. 103–104. ISSN: 1094-3641. DOI: 10.1145/2692956.2663188.

[5] K. Rustan M. Leino. "Dafny: An automatic program verifier for functional correctness". In: *International conference on logic for programming artificial intelligence and reasoning*. Springer. 2010, pp. 348–370.