Advanced Logical Proofs in a Verifier

Bachelor's Project Description

Dina Weiersmüller
Supervised by Linard Arquint, Thibault Dardinier,
and Prof. Peter Müller

September 2022

1 Introduction and Motivation

Viper [1] is an intermediate verification language with back-ends that verify whether a Viper program is correct. A Viper program is correct if all its methods are partially correct. A method is partially correct with respect to a precondition and a postcondition iff the resulting state after executing the method satisfies the postcondition given that the initial state satisfies the method's precondition. A program is verified by Viper in a modular way, which means that the program is split into methods that are verified separately. Viper also includes other specifications, for instance to specify loop invariants.
In general, Viper is capable of verifying non-trivial programs such as programs modifying the heap. Thus, Viper is being used by several front-ends to verify programs written in Go [2], Rust [3], Java [4], and Python [5]. In Viper, more complex properties can be expressed and verified as methods, but the proof code is restricted to the language features Viper offers. Particularly relevant to this thesis, Viper currently does not offer existential elimination and universal introduction, i.e. the introduction of the universal quantifier and the elimination of the existential quantifier. Additionally Viper currently has no support for lemmas instead lemmas have to be simulated with methods. Past projects had to introduce local assumptions to work around these missing language features. Therefore, the main goal of this thesis is to extend Viper in a way that such local assumptions can be avoided and existential elimination, universal introduction and lemmas can be used in a sound way.

1.1 Motivating Example

To prove an algorithm correct by induction one has to find some sort of inductive argument. In the case of quicksort [6], this argument is performed on the subarrays that are created by splitting the input array on a pivot. One subarray consists of elements smaller than the pivot, the other consists of elements greater than the pivot. The base case, sorting a singleton array, is trivial. In the inductive step the array of length n+1 is sorted by performing a recursive call on the subarrays with length $\leq n$, where we can use the induction hypothesis. In other cases, it might not be as easy to find a suitable induction hypothesis. One such example is shearsort [7]. Shearsort takes as input an $n \times n$-matrix

and outputs the matrix sorted in a snake-like order in $\lceil \log_2(n) \rceil + 1$ iterations. Each iteration consists of two main steps:

Algorithm 1 Shearsort

```
1: repeat ⌈log2(n)⌉ + 1 times:
2:     Sort rows in an alternating way.
3:         First row: left-to-right
4:         Second row: right-to-left
5:         ...
6:     Sort columns top-to-bottom.
```

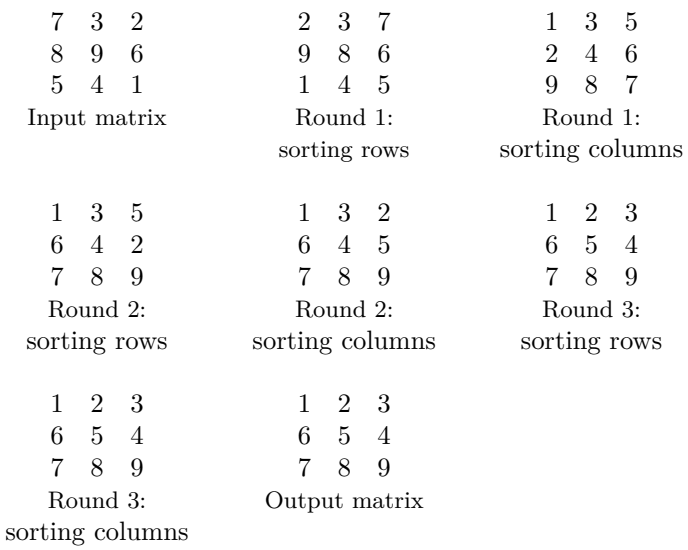

Figure 1: Shearsort algorithm executed on a 3×3-matrix.

Directly expressing a suitable induction hypothesis can be challenging but we can apply the 0-1-principle [8] as an intermediate step to simplify the problem.

1.2 0-1-Principle

The 0-1 sorting principle informally states that if a sorting algorithm sorts all inputs of 0's and 1's , then it sorts any input. This principle can be applied in Viper to prove the correctness of shearsort as follows. First an integer threshold t has to be fixed, then any value $\leq t$ can be viewed as 0, and values $> t$ as 1. Using Viper and this fixed threshold t, it can now be proven that the created 0-1 matrix is sorted after the algorithm has been executed, as shown in the example below. In the last step using universal introduction, it can be proven that the 0-1 matrix will be sorted for any threshold t.

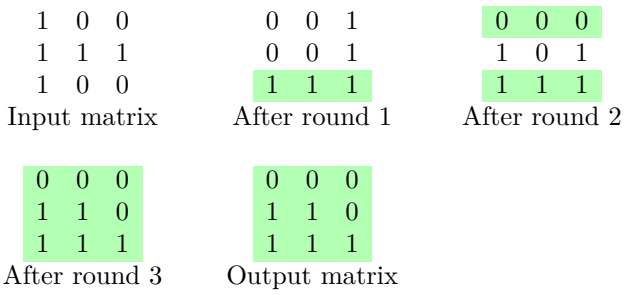

Figure 2: Shearsort algorithm executed on a 3×3-matrix using the 0-1-principle with threshold $t = 4$.

In the example above, the rows that are marked with green are correctly sorted. It is now clearly visible that after each round the matrix is "more sorted".

2 Useful Reasoning Features in Viper

2.1 Existential Elimination

Existential elimination means that if it can be proven that $\exists x.P(x)$ holds, then the existential quantifier can be removed and a witness w can be obtained such that $P(w)$ holds. Naively adding existential elimination to Viper, encoded as the following Viper statement, would be unsound.

```
⟦ obtain x::T where P(x) ⟧  →  var w: T;
                               assume P(w)
```

Figure 3: An unsound Viper encoding for existential elimination.

To correct this encoding, we first need to assert that there exists a variable that satisfies P. Such a correct encoding would be:

```
⟦ obtain x::T where P(x) ⟧  →  assert exists x: T :: P(x);
                               var w: T;
                               assume P(w)
```

Figure 4: A correct Viper encoding for existential elimination.

The assert statement ensures that a witness is guaranteed to exist.

2.2 Universal Introduction

After proving that $P(x)$ holds for an arbitrary variable x, universal introduction can be applied to obtain $\forall x.P(x)$. In general we would like to be able to prove that $\forall k.P(k) \implies Q(k)$ in Viper. One seemingly possible encoding would be:

```
⌈ prove forall k:T assuming P(k) implies Q(k) {      ⌉
⌊     <code provided by user>                        ⌋
  }
```

```
  var arb_k: T;
  assume P(arb_k);
→ <code provided by user>
  assert Q(arb_k);
  assume forall k:T :: P(k) ==> Q(k)
```

Figure 5: An unsound Viper encoding for universal introduction.

Even though this seems to work at first glance, there are some problems with this code. For instance, the assumption on k needs to be guarded by a boolean value, since if $P(k)$ is not satisfiable, then `assume P(k)` kills all traces and the rest of the program trivially verifies, even though it might be incorrect. An if statement including an arbitrary boolean value ensures that the rest of the program is still checked. Moreover, a label has to be added such that we can use $P(k)$ in the forall assumption in the state it had before the provided code was executed.

```
⌈ prove forall k:T assuming P(k) implies Q(k) {      ⌉
⌊     <code provided by user>                        ⌋
  }
```

```
  var b: Bool;
  var arb_k: T;
  label l;
  if(b) {
→     assume P(arb_k)
  }
  <code provided by user>
  assert b ==> Q(arb_k);
  assume forall k:T :: old[l](P(k)) ==> Q(k)
```

Figure 6: An improved Viper encoding for universal introduction.

It also has to be analysed whether in the provided code $k :: T$ influences any other variables in the program, for example in an assignment or a conditional assignment. In the example below $P(k)$ is `true` und $Q(k)$ is the assertion `x == k`.

```
var x: T;
prove forall k:T assuming true implies x == k {
    x := k
}
```

Figure 7: Example of an assignment where variable k influences x.

Additionally, P and Q have to be restricted, for example it is required that they are expressions.

2.3 Pure Method as Lemma

In order to prove an implication, like in the encoding for the universal introduction, one might want to use a lemma. Currently Viper does not offer support for lemmas and therefore the user has to simulate lemmas with methods. However, this is prone to errors. An example of such an incorrect proof is the following code.

```
method incorrect_proof(n: Int)
    ensures false
{
    incorrect_proof(n-1)
}
```

Figure 8: Incorrect induction proof

This proof is not guaranteed to terminate which is equivalent to the induction proof not having a base case making it incorrect. Therefore, we would like to add sound support for lemmas, for example by checking whether the lemma terminates. Furthermore, lemmas should have no side-effects, i.e. no write operations to the heap, such that they might for example be called in an old context.

3 Goals

3.1 Core Goals

3.1.1 Existential Elimination

- Design a Viper-to-Viper plug-in and add syntax for existential elimination like in Section 2.1.
- Implement the plug-in performing the corresponding encoding.
- Write test cases for the plug-in.

3.1.2 Universal Introduction

- Design a Viper-to-Viper plug-in and add syntax for universal introduction, see Section 2.2.
- Develop a modular static analysis to make sure the universal introduction is safe and implement it.
- Add the encoding for universal introduction to the plug-in.
- Write test cases for the universal introduction in the plug-in.

3.1.3 Lemmas

- Add support for lemmas in the Viper-to-Viper plug-in, see Section 2.3.
- Write test cases for lemmas in the plug-in.

3.1.4 Case Studies/Evaluation:

- Evaluate the Viper plug-in by verifying one or more sorting algorithms with the 0-1-principle.
- Evaluate the Viper plug-in on an existing Viper codebase and qualitatively asses the expressiveness of the plug-in compared to the local assumptions in the codebase.

3.2 Extension Goals

- Add support for simple set, sequence and map comprehensions as part of the plug-in.
- Add support for the new features in Gobra

References

[1] Peter Müller, Malte Schwerhoff, and Alexander J. Summers. "Viper: A Verification Infrastructure for Permission-Based Reasoning". In: *Verification, Model Checking, and Abstract Interpretation*. Ed. by Barbara Jobstmann and K. Rustan M. Leino. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 41–62. ISBN: 978-3-662-49122-5.

[2] Felix A. Wolf et al. "Gobra: Modular Specification and Verification of Go Programs". In: *Computer Aided Verification*. Ed. by Alexandra Silva and K. Rustan M. Leino. Cham: Springer International Publishing, 2021, pp. 367–379. ISBN: 978-3-030-81685-8.

[3] Vytautas Astrauskas et al. "Leveraging Rust Types for Modular Specification and Verification". In: *Proc. ACM Program. Lang.* 3.OOPSLA (Oct. 2019). DOI: 10.1145/3360573. URL: https://doi.org/10.1145/3360573.

[4] Stefan Blom et al. "The VerCors Tool Set: Verification of Parallel and Concurrent Software". In: *Integrated Formal Methods*. Ed. by Nadia Polikarpova and Steve Schneider. Cham: Springer International Publishing, 2017, pp. 102–110. ISBN: 978-3-319-66845-1.

[5] Marco Eilers and Peter Müller. "Nagini: A Static Verifier for Python". In: *Computer Aided Verification*. Ed. by Hana Chockler and Georg Weissenbacher. Cham: Springer International Publishing, 2018, pp. 596–603. ISBN: 978-3-319-96145-3.

[6] C. A. R. Hoare. "Quicksort". In: *The Computer Journal* 5.1 (Jan. 1962), pp. 10–16. ISSN: 0010-4620. DOI: 10.1093/comjnl/5.1.10. eprint: https://academic.oup.com/comjnl/article-pdf/5/1/10/1111445/050010.pdf. URL: https://doi.org/10.1093/comjnl/5.1.10.

[7] I.D. Scherson and S. Sen. “Parallel sorting in two-dimensional VLSI models of computation”. In: *IEEE Transactions on Computers* 38.2 (1989), pp. 238–249. DOI: 10.1109/12.16500.

[8] Donald E. Knuth. *The Art of Computer Programming, Volume 3: (2nd Ed.) Sorting and Searching*. USA: Addison Wesley Longman Publishing Co., Inc., 1998. ISBN: 0201896850.