Improving the Support for Magic Wands in Viper

Practical Work Project Description

TODOR HRISTOV

Supervisor: Thibault Dardinier

1 INTRODUCTION

Viper. Viper (a verification infrastructure for permission-based reasoning) [8] is a malleable and expressive verification language and tool chain which can be used for building verification front-ends for numerous other languages with diverse features. Those front-ends include Nagini for Python [6], Prusti for Rust [1] and Gobra for Go [14]. The Viper tool chain uses two different backends: Silicon and Carbon. Silicon is based on symbolic execution, while Carbon uses an additional translation to another intermediate verification language — Boogie [9], which can then be automatically verified using the weakest precondition.

Separation Logic. Viper is based on Separation Logic [11], which is an extension of Hoare logic that permits reasoning about mutable heap locations that may be shared. During verification, the program state is represented by a heap, which is a map from a reference (similar to a pointer in C++) and field to a permission amount, quantified by a rational number between 0 and 1 (both included). In the simplest case, *acc(x.f)* can be used to represent the full permission (equal to 1.0) for the field f of reference x. Having full permission allows us to have a read/write access to a heap location. Viper also supports fractional permissions [2], with the idea being that any level of permission $p \in (0, 1)$ states that we have read access to a heap location. Furthermore, because we have a non-zero permission to the heap location, we can be sure than no other process has write permission. This ensures that this heap location cannot be modified by other concurrent processes.

Data Structures. To represent more complex permissions in Viper, we can use *predicates* and *magic wands*. Predicates are used when we want to define a recursive data structure. For example, a binary tree structure contains either a null or another binary tree as each of its children. This can be expressed as:

```
1  predicate bt(this: Ref) {
2      acc(this.left) &&
3      acc(this.right) &&
4      acc(this.value) &&
5      (this.left  != null ==> bt(this.left)) &&
6      (this.right != null ==> bt(this.right))
7  }
```

Magic Wands. Another non-trivial type of permissions in Viper is the so-called *magic wand*. It can be used to represent the leftover permission after a given set of permissions is taken from a predicate. Figure 1 shows an example of a simple magic wand.

Usages. Magic Wands are usually used for loop invariants, where we want to record the permissions of the data structure while we are traversing it. Another natural occurrence is in the postcondition of methods. For example, using the example on Figure 1, the magic wand can be returned in addition to *Tree(y)*, so that the client can recover *Tree(x)*.

Automatic Verification. Using magic wands in automatic verifiers is inherently hard, as they are undecidable [3]. Their complexity has led to unsound implementations in the past [12], which further points to their complexity.

Author's address: Todor Hristov, thristov@ethz.ch.

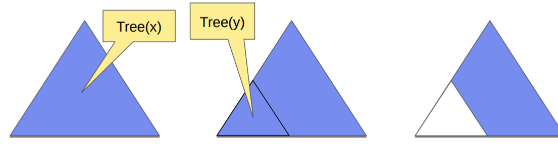

Fig. 1. The leftmost triangle shows in blue what are the permissions in the predicate *Tree(x)*. The triangle in the middle shows the subset of *Tree(x)* which correspond to the *Tree(y)* predicate. The blue coloured triangle part on the right corresponds to the permissions of *Tree(x)* after we have removed the permissions to *Tree(y)*. Those permissions can be concisely represented by the magic wand $(Tree(y) -\star Tree(x))$ [13].

Sound Algorithm. There has been recent advancement in the automation of magic wands. Most notable, a new sound approach to their automation in the Carbon backend has been developed [5]. In order to correctly use magic wands, we need to be able to remove permissions from our heap and "package" it inside a wand. This "packaging" operation can choose to remove different amounts of permissions from the heap. For example, if it chooses to remove too much the verifier may become incomplete, while removing too little may result in unsoundness. The permissions, which we remove from the heap in order to create the magic wand, is called a *footprint*.

2 PROJECT GOALS

The aim of this project is to improve the support for *magic wands* in Viper. More precisely, it focuses on improving the *footprint* computation of the *magic wands* in Carbon. This is needed, as the implementation currently in place is incomplete, and will result in falsely rejecting programs that are correct. Improving the *footprint* computation will allow it to be used in more complex and diverse applications, which will improve its adoption without impeding its soundness.

3 TASKS

The stated goals can be achieved by the following steps:

(1) Improve the heuristics used to calculate the amount of permissions we need to take from the heap when packaging a magic wand (i.e. when calculating the footprint of the magic wand):
   (a) Change the use of minimum to infimum when calculating the footprint.
   The difference between the two approaches arises when we have multiple requirements on the amount of permissions we need to take from the heap. This can be shown by the following examples:
   - Example 1:
     For field $x.f$ we have the following requirements for the permission p we take from the environment:
     $$p \geq \frac{1}{2}, p \geq \frac{1}{3}, p \geq \frac{1}{4}, p \geq \frac{1}{5}, p \geq \frac{1}{6}, \ldots$$
     Currently, the minimum condition used to calculate what is the value of p states that p is equal to one of the aforementioned requirements:
     $$p = \frac{1}{2} \vee p = \frac{1}{3} \vee p = \frac{1}{4} \vee p = \frac{1}{5} \vee p = \frac{1}{6}, \ldots$$
     For this example, this way of calculation is correct, as it yields $p = \frac{1}{2}$ successfully.
   - Example 2:

If we have the following requirements:

$$p \geq \frac{1}{2}, p \geq \frac{2}{3}, p \geq \frac{3}{4}, p \geq \frac{4}{5}, p \geq \frac{5}{6}, \ldots$$

The infinite series goes to 1, which is not part of the set itself. Therefore we cannot say that the upper bound is part of the set. What we do instead is:

$$\forall b \in \mathbb{R}, (b \geq \frac{1}{2} \land b \geq \frac{2}{3} \land b \geq \frac{3}{4} \land b \geq \frac{4}{5} \land b \geq \frac{5}{6} \land \ldots) \Rightarrow (p \leq b)$$

Which correctly gives $p = 1$.

The new infimum way of calculating the upper bounds also works in other edge cases, such as when the set of lower-bound requirements is empty.

(b) Compare the new infimum implementation with the current solution.

(c) If the new implementation is worse, try to improve the current minimum computation.

(2) Support for Quantified Permissions [10].

(3) Improve the footprint algorithm with one or more of the following:
- Different strategies for packaging.
- Nested package statements.
- Known-folded permissions [7].
- Add support for fractional magic wands [4].

Possible Extensions.

The following list contains improvements that can be additionally incorporated if the project timeline allows:

(1) Exploring the notion of *Generalised magic wands*. This generalises the ordinary magic wand, which consists of a single assertion on its left and right side, to multiple assertions with arbitrary connections between them. This additional expressiveness will allow the representation of additional complex data structures and concurrent programming paradigms.

REFERENCES

[1] V. Astrauskas, A. Bílý, J. Fiala, Z. Grannan, C. Matheja, P. Müller, F. Poli, and A. J. Summers. 2022. The Prusti Project: Formal Verification for Rust (invited). In *NASA Formal Methods (14th International Symposium)*. Springer, 88–108. https://link.springer.com/chapter/10.1007/978-3-031-06773-0_5

[2] John Boyland. 2003. Checking interference with fractional permissions. In *International Static Analysis Symposium*. Springer, 55–72.

[3] Rémi Brochenin, Stéphane Demri, and Etienne Lozes. 2012. On the almighty wand. *Information and Computation* 211 (2012), 106–137.

[4] Thibault Dardinier, Peter Müller, and Alexanders J. Summers. 2022. Fractional Resources in Unbounded Separation Logic. *Proc. ACM Program. Lang.* 6, OOPSLA2, Article 163 (oct 2022), 27 pages. https://doi.org/10.1145/3563326

[5] Thibault Dardinier, Gaurav Parthasarathy, Noé Weeks, Peter Müller, and Alexander J Summers. 2022. Sound Automation of Magic Wands. In *International Conference on Computer Aided Verification*. Springer, 130–151.

[6] Marco Eilers and Peter Müller. 2018. Nagini: a static verifier for Python. In *International Conference on Computer Aided Verification*. Springer, 596–603.

[7] Stefan Heule, Ioannis T Kassios, Peter Müller, and Alexander J Summers. 2013. Verification condition generation for permission logics with abstract predicates and abstraction functions. In *European Conference on Object-Oriented Programming*. Springer, 451–476.

[8] U. Juhasz, I. T. Kassios, P. Müller, M. Novacek, M. Schwerhoff, and A. J. Summers. 2014. *Viper: A Verification Infrastructure for Permission-Based Reasoning*. Technical Report. ETH Zurich.

[9] K. Rustan M. Leino. 2008. This is Boogie 2. (June 2008). https://www.microsoft.com/en-us/research/publication/this-is-boogie-2-2/

[10] Peter Müller, Malte Schwerhoff, and Alexander J Summers. 2016. Automatic verification of iterated separating conjunctions using symbolic execution. In *International Conference on Computer Aided Verification*. Springer, 405–425.

[11] John C Reynolds. 2002. Separation logic: A logic for shared mutable data structures. In *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*. IEEE, 55–74.

[12] Malte Schwerhoff and Alexander J Summers. 2015. Lightweight suppoert for magic wands in an automatic verifier. In *29th European Conference on Object-Oriented Programming (ECOOP 2015)*, Vol. 37. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 614–638.

[13] Thibault Dardinier, Gaurav Parthasarathy, Noé Weeks, Peter Müller, and Alexander J. Summers. [n. d.]. Sound Automation of Magic Wands. https://easychair.org/smart-slide/slide/45nV#. [Online; accessed 08-September-2022].

[14] F. A. Wolf, L. Arquint, M. Clochard, W. Oortwijn, J. C. Pereira, and P. Müller. 2021. Gobra: Modular Specification and Verification of Go Programs. In *Computer Aided Verification (CAV) (LNCS, Vol. 12759)*, Alexandra Silva and K. Rustan M. Leino (Eds.). Springer International Publishing, 367–379. https://link.springer.com/chapter/10.1007/978-3-030-81685-8_17