Adding Native Support for Havoc in Viper

Practical Work Project Description

Daniel Zhang
danzhang@student.ethz.ch
Supervised by Dr. Malte Schwerhoff
Department of Computer Science
ETH Zürich, Switzerland

1 Introduction

Viper Viper, or Verification Infrastructure for Permission-Based Reasoning, is a tool chain and intermediate language used for program verification [3]. The Viper language provides a common interface for several front-ends, which verify code in languages such as Go [5] and Rust [1]. Viper has two implementations: Silicon, a verifier based on symbolic sxecution [4]; and Carbon, based on verification condition generation via translation to Boogie (a simpler intermediate verification language [2]).

The Viper language is based on separation logic, an extension of Hoare logic, which is especially useful for verifying programs with mutable heap state, and concurrency. At any given point in the program's verification, the heap is represented as a set of *resources* (and constraints on their values), and a resource may only be accessed if the necessary permission are currently available. The most basic resource is a *heap location*, denoted by $\texttt{acc}(e.f)$. In this case, e denotes an object with field f, and the built-in `acc` predicate denotes that the user has write access to it. Viper also includes two other kinds of resources: *predicates* (for specifying recursive structures) and *magic wands* (for specifying decomposed structures). In addition, *quantified permissions* allow users to specify ownership of an unbounded number of resources, e.g. for non-recursive data structures. For example, if S is a set of references, then a quantified permission assertion could describe access to the f-field of all objects in S.

Havoc When encoding the semantics of a source program (e.g. in Go) in Viper, it is often necessary to remove all prior knowledge about a resource. For example, when modelling potential environment interference from other threads. For local variables, this is commonly known as *havocking*. In Viper, this can be encoded by temporarily losing, and then regaining, access to resources.

However, this encoding has two disadvantages: the corresponding Viper code must be generated repeatedly and by all front-ends, and we have experimental evidence suggesting that a native support for havocking will be much more efficient in certain situations. The Viper team therefore decided to add native support for havocking to the language.

2 Project Goal

The goal of this project is to add a suitable `havoc` statement to Viper, that takes the resource to havoc as an argument. From the user's perspective, the `havoc` statement will update the resource's underlying memory to an unknown state (e.g. value for a single heap location). The syntax and behavior of `havoc` has already been outlined by the Programming Methodology Group – in this project, we aim to formalize and implement the behavior in the Silicon verifier. Moreover, an associated `havocall` statement will be added to support havocking quantified resources.

3 Tasks

The above project goal can be divided into the following tasks:

1. Implement the `havoc` statement for field resources:
   a) Add the proposed syntax to Viper's parser and type checker, which is shared by both Silicon and Carbon.
   b) Handle `havoc` statements appropriately in Silicon.
   c) Benchmark the performance of encoded vs. native havocking.
2. Repeat the above steps for predicates and magic wands.
3. Repeat the above steps for `havocall` and quantified resources.

A possible extension goal would be to to implement `havoc` and `havocall` in Carbon.

References

[1] Vytautas Astrauskas, Peter Müller, and Alexander J. Summers Frederico Poli. Leveraging rust types for modular specification and verification. *Proceedings of the ACM on Programming Languages*, 3, 2019.

[2] K. Rustan M. Leino. This is boogie 2, 2008.

[3] Alex J. Summers Peter Müller, Malte Schwerhoff. Viper: A verification infrastructure for permission-based reasoning. *International Conference on Verification, Model Checking, and Abstract Interpretation*, pages 41–62, 2016.

[4] Malte Hermann Schwerhoff. *Advancing Automated, Permission-Based Program Verifcation Using Symbolic Execution*. PhD thesis, ETH Zürich, 2016.

[5] Felix A. Wolf, Linard Arquint, Martin Clochard, Wytse Oortwijn, João C. Pereira, and Peter Müller. Gobra: Modular specification and verification of go programs. *Computer Aided Verification*, 2021.