Foundational Proofs for Automated Verifiers with Separation Logic Predicates

Master's Thesis Project Description

Yushuo Xiao

Supervised by Thibault Dardinier, Gaurav Parthasarathy, and Prof. Dr. Peter Müller

1 Introduction

Automated program verifiers (or program verifiers in short) are tools that take as input a program, a formal specification, and potential annotations, and try to automatically establish that the program satisfies the specification, or point out the part of the program that may have an error. A program verifier should be sound, in the sense that the verifier reports success only if the input program is correct. However, the soundness of program verifiers themselves is usually unverified. Most program verifiers (such as Why3, Dafny, and Viper [3,4,5]) are implemented in complex high-level languages, have large implementations, and are subject to rapid changes. As a result, bugs can and do arise in these implementations, and thus we want an approach to formally ensure that program verifiers are sound.

Many program verifiers are *translational program verifiers* that translate a front-end program into some intermediate verification language (IVL). An IVL comes with its own back-end verifier that translates the IVL program into a set of formulas whose validity is checked using an SMT solver. Soundness of such a program verifier can be guaranteed by establishing the soundness of each translation: (1) Front-end soundness: the correctness of the IVL program implies the correctness of the front-end program, (2) IVL back-end soundness: validity of the formulas implies correctness of the IVL program.

Recent work by Parthasarathy et al. [1] offers an approach to formally establish the front-end soundness for translational program verifier implementations. Proving the correctness of such implementations once and for all is practically infeasible, since these implementations are large and are usually written in languages that lack a formalization. Parthasarathy et al. instead propose to instrument the existing implementation to generate a *certificate* every time the implementation is run. They apply their approach to one of the Viper [5] verifiers (a verifier based on separation logic [10]), which translates input Viper programs into the Boogie IVL [6] (the Boogie verifier then translates the Boogie program to a set of formulas). That is, their instrumented version of the Viper-to-Boogie translation produces an Isabelle [7] proof stating that if the Boogie program, emitted on that particular run, is correct, then the source Viper program is also correct w.r.t. the Viper semantics.

This proof generation method, along with the authors' work on generating Boogie-to-SMT certificates [2], gives strong guarantees of the correctness of a successful Viper verifier invocation. This methodology shows promising results and feasibility of the general approach. However, the current implementation covers only a core subset of the Viper language, which in particular does not include separation logic *predicates* and heap-dependent *functions*. In Viper, predicates are the main mechanism for abstracting separation logic resources [9], and the only direct way to describe ownership of recursive data structures,

such as lists and trees. Functions are pure procedures to describe functional properties of these data structures (e.g., the length or the elements in a linked list). These two features are used in realistic program verification ubiquitously. Therefore, it is essential to add support for these features to the proof generation if we want the proof generation also to cover the correctness of a large class of real-world programs.

This project aims to extend the current Viper-to-Boogie proof generation framework to support (recursively-defined) predicates and optionally functions. A brief description of the existing framework and potential challenges are presented in Section 2. The goals and the concrete work to be carried out are outlined in Section 3.

2 Approach and Challenges

The project is based on the Viper-to-Boogie translator, the verification-condition-generation-based back-end of Viper. The translation and verification framework and relevant semantics for proof generation are illustrated in Figure 1.

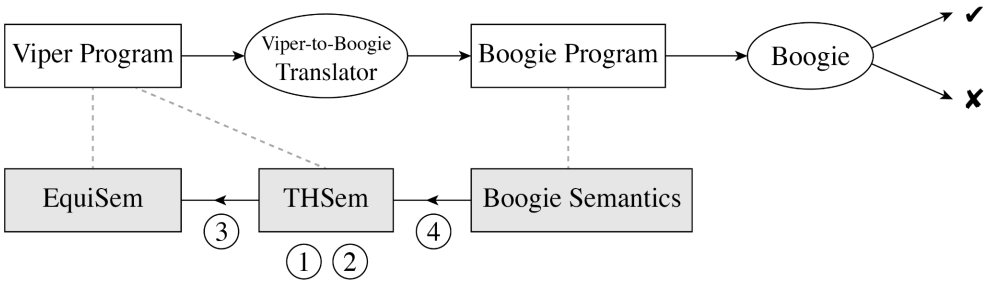

Figure 1. The Viper-to-Boogie translator and its proof generation framework.

The upper half of Figure 1 illustrates how the Viper-to-Boogie translation and verification work as explained in the introduction. The lower half of the figure illustrates several semantics that are used to justify the correctness of a Viper program translation. They are already partially formalized in Isabelle. THSem is a Viper semantics that reflects how predicates are treated operationally in Viper. It is the semantics that is used by the proof-producing Viper-to-Boogie implementation (which does not yet support predicates itself). EquiSem on the left intends to be a more high-level and intuitive semantics for Viper programs. The Boogie semantics on the right describes the expected behavior of a Boogie program. We introduce THSem and EquiSem in more detail below.

THSem (*total heap semantics*) is a Viper semantics designed to reflect how (recursive) predicates are treated in Viper. Viper (and in general other separation-logic-based verifiers) treats Viper predicates *isorecursively*: a predicate instance is differentiated from its body, and a conversion between the two is achieved by explicit fold and unfold statements (possibly through heuristics-based inference). To reflect this, THSem also treats a predicate instance and its body differently in the program state. However, treating predicate instances and the permissions in their bodies differently complicates the design of

THSem. The disparity is reflected by a total heap semantics design, in the sense that information can still be stored in the state even if it holds no permission to the location. The total heap design and the coexistence of predicates and memory locations requires a notion of *state consistency*, since we need to exclude any states that are not consistent under any unfolding operations (that is, operations that replace a predicate instance by the resources in its body). In summary, we have to account for these challenges in this project and formalize the correct semantics of predicates in THSem (not all parts related to predicates have been formalized in THSem yet).

EquiSem (*equirecursive semantics*) is a more high-level Viper semantics to reflect the intuitive meaning of a predicate. In this semantics, a predicate instance is treated as equivalent to its body, using the least fixed-point interpretation. Therefore, the effects of fold and unfold statements in Viper source programs are completely nullified. That is, they can be removed from the program without affecting the program behavior. As opposed to THSem, the program state in EquiSem is a partial heap, as is standard in separation logic.

There are two important goals in this project. The first goal is to prove that if the Boogie program is correct, then the Viper program, possibly containing predicates, is correct w.r.t. THSem, which we will achieve by extending the proof-producing instrumentation of the Viper-to-Boogie implementation. The second goal is to increase the confidence that THSem describes a meaningful semantics for predicates by showing that a correct Viper program w.r.t. THSem implies correctness w.r.t. EquiSem. The tasks of this project mainly concern the first goal, but also touch the second goal (i.e. connection to EquiSem) as we explain in the next section.

3 Goals

The four core goals are marked with circled numbers in Figure 1 to illustrate which part they concern.

3.1 Core Goals

1. Define and formalize (in Isabelle) the semantics for predicates in THSem, supporting fold and unfold statements and unfolding expressions with fractional permissions [8]. This will require adjusting the semantics of exhale to also support predicates and defining a notion of consistency for THSem states.
2. Design an approach to extend the semantics from core goal 1 to functions. Since functions also need to be supported in this framework eventually (potentially outside of this project) and they interact with predicates subtly, we need to be careful in our design of the semantics of predicates and consider the possibility of supporting functions later on.
3. Connect the extended THSem from core goal 1 to EquiSem by connecting THSem states to EquiSem states. This will increase the confidence in our extension to THSem. Doing a full proof connecting THSem and EquiSem requires significant effort which might not be possible under the time constraint of this project, so we will only try to connect THSem states to EquiSem states, which constitutes the basis of a THSem-to-EquiSem proof.

4. Extend the Viper-to-Boogie proof generation to support predicates, which on every run of the implementation automatically generates an Isabelle proof showing that if a Boogie program is correct, then so is the Viper program w.r.t. THSem. This will be an extension to the existing proof-producing Viper-to-Boogie implementation [1].

3.2 Extension Goals

1. Further extend the semantics and the proof generation framework to support wildcard permissions, repeating the core goals for these features.
2. Formalize the semantics of functions, and repeat the core goals 3 and 4 for functions.

References

[1] Gaurav Parthasarathy, Thibault Dardinier, Benjamin Bonneau, Peter Müller, and Alexander J. Summers. 2024. Towards Trustworthy Automated Program Verifiers: Formally Validating Translations into an Intermediate Verification Language (to appear at PLDI 2024). https://arxiv.org/abs/2404.03614

[2] Gaurav Parthasarathy, Peter Müller, Alexander J. Summers. 2021. Formally Validating a Practical Verification Condition Generator. In: Silva, A., Leino, K.R.M. (Eds.) Computer Aided Verification. CAV 2021. Lecture Notes in Computer Science, Vol. 12760. Springer, Cham. https://doi.org/10.1007/978-3-030-81688-9_33

[3] Jean-Christophe Filliâtre and Andrei Paskevich. 2013. Why3 — Where Programs Meet Provers. In European Symposium on Programming (ESOP), Matthias Felleisen and Philippa Gardner (Eds.). https://doi.org/10.1007/978-3-642-37036-6_8

[4] K. Rustan M. Leino. 2010. Dafny: An Automatic Program Verifier for Functional Correctness. In Logic for Programming, Artificial Intelligence, and Reasoning (LPAR), Edmund M. Clarke and Andrei Voronkov (Eds.). https://doi.org/10.1007/978-3-642-17511-4_20

[5] Peter Müller, Malte Schwerhoff, and Alexander J. Summers. 2016. Viper: A Verification Infrastructure for Permission-Based Reasoning. In Verification, Model Checking, and Abstract Interpretation (VMCAI), Barbara Jobstmann and K. Rustan M. Leino (Eds.). https://doi.org/10.1007/978-3-662-49122-5_2

[6] K. Rustan M. Leino. 2008. This is Boogie 2. (2008). Available from http://research.microsoft.com/en-us/um/people/leino/papers/krml178.pdf

[7] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. 2002. Isabelle/HOL - A Proof Assistant for Higher-Order Logic. Lecture Notes in Computer Science, Vol. 2283. Springer. https://doi.org/10.1007/3-540-45949-9

[8] Thibault Dardinier, Peter Müller, and Alexander J. Summers. 2022. Fractional resources in unbounded separation logic. Proc. ACM Program. Lang. 6, OOPSLA2, Article 163 (October 2022), 27 pages. https://doi.org/10.1145/3563326

[9] Stefan Heule, Ioannis T. Kassios, Peter Müller, Alexander J. Summers. 2013. Verification Condition Generation for Permission Logics with Abstract Predicates and Abstraction Functions. In: Castagna, G. (Eds.). ECOOP 2013 – Object-Oriented Programming. ECOOP 2013. Lecture Notes in Computer Science, Vol. 7920. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-39038-8_19

[10] John C. Reynolds. 2002. Separation Logic: A Logic for Shared Mutable Data Structures. Logic in Computer Science (LICS), 55–74. https://doi.org/10.1109/LICS.2002.1029817