Code Reachability and Soundness Verification Using Refute in Prusti

Practical Work Project Description

Šimon Hrabec

February 10, 2023

1 Introduction

Prusti and Viper Prusti [1] is a Rust verifier built on top of the Viper [4] verification infrastructure. Besides Prusti there exist front-ends for other languages such as Gobra [5] (Go language) and Nagini [2] (Python). The Viper infrastructure uses one of two verification back-ends - Silicon (based on symbolic execution) or Carbon (based on verification condition generation). The back-end operates on the Viper intermediate verification language that is generated by Prusti.

Adding refute statements to Prusti One of the basic features of the Viper intermediate verification language (IVL) is the assert statement. Viper also supports its counterpart - the refute statement. The assert statement ensures that this condition holds in any program trace. The refute statement shows that the condition is not valid for all execution paths - It fails if the condition holds for all possible traces. This project proposes to use the Viper refute statement in pursuit of two goals - locating dead (unreachable) code and detecting unsound reasoning.

Refute implementation Viper internally converts refute statements into assert statements and then inverts the results - it treats errors as correct behavior and issues an error for passing assert statements. This approach is taken because the refute statement and a simple negation of the assert condition are not equivalent.

2 Project goals

2.1 Detecting unreachable code

Unreachable code is not desirable for multiple reasons. It can unnecessarily enlarge the target binary and possibly reduce performance (due to an increase in instruction cache misses). It muddies the program (having no function, programmers might still read it) and most importantly the code might have a function but its unreachability is caused by mistake. In case of the presence of a dead code, it is desirable that the developer is notified with a warning. Modern compilers already have mechanisms to do so and the following code snippets would generate a warning:

```rust
pub fn square(num: i32) -> i32 {
    return 0;
    let result = num * num;
    return result;
}
```

However, there are many examples where the compiler fails to detect the issue. The previous example had code after a return statement, which does not require any reasoning. Another way to get unreachable code is to have a branch that gets executed based on a condition that is not satisfiable. The next example shows a sign function where the accidental use of non-strict comparisons leads to incorrect behavior. This bug also manifests itself by making the third branch unreachable. However, compilers are generally limited in their abilities to detect cases of unreachable code that require condition satisfiability check and the Rust compiler does not issue a warning for this function.

```rust
fn sign(val: i32) -> i32 {
    if val <= 0 {
        -1
    } else if val >= 0 {
        1
    } else {
        0
    }
}
```

The convenient implementation of pattern matching in Rust is also a way of branching. When also used with match guards (added if condition) the compiler might fail to flag an arm as unreachable for the same reasons.

Perhaps the simplest way to write code that is not reachable is to have an outright contradictory condition.

```rust
pub fn square(num: i32, cond: bool) -> i32 {
    if cond && !cond {
        let result = num * num;
        return result;
    }
    return 0;
}
```

This project proposes to use the refute statement to detect unreachable code. Placing a `refute false` statement into a dead code will lead to a verification failure - if the code is not reachable then there is no trace passing through this branch. In this case false can be proven which makes the refute statement fail and issue an error. The general approach might be to place this statement at the beginning of every basic block that corresponds to a user code.

2.2 Detecting unsoundness

The verification process relies on Prusti-generated axioms and on user-provided preconditions, postconditions, assumptions, and invariants. Mistakes in this part can introduce unsoundness and the code will verify regardless of its correctness. It is rather simple to achieve such a state. It can be done in several ways:

1. Assume statement - By inserting a contradiction such as an incorrect formulation of Fermat's last theorem: `assume x*x*x + y*y*y != z*z*z`
2. Abstract methods - Viper allows the use of a method without a body. In this case, the specifications are not checked and for some input, the postconditions might be contradictory.
3. Axioms - Similarly to abstract methods, unsoundness can be introduced by improper axioms.

Unsoundness can be detected by manually inserting `assert false` into the Viper code, which should fail the verification under normal circumstances. Successful verification then shows the presence of unsound reasoning (invalid axiom, assumption, or abstract method postcondition). However, this approach is tedious and not systematic. Hence, it is desirable to perform a similar action without the user's random trials. Inserting `refute false` statements into the code can then verify the absence of unsoundness.

An issue arises as we try to use the same construct to detect different things - we need a way to distinguish when are we dealing with unreachable code and when with unsoundness. The proposed solution of the two features intends to place the `refute false` statements in different locations. In case of an unreachable code the refute statement will fail at the beginning of the branch, but it should pass in places before it. Similarly for unsoundness the refute statement should pass before the place of interest (assume statement or abstract method call) but fail right after it. If we write an incorrect assume statement at the beginning of a function all the refute statements inside branches will fail, but we should not interpret those failures as unreachable code.

3 Tasks

- Add support to Prusti for `prusti_refute!` macro. Prusti already has support for `prusti_assert!` and `prusti_assume!` macros [3]. This allows users to write Viper statements (assume/assert) directly into Rust code that only exist in Viper for verification purposes and do not affect the final binary. Implement a `prusti_refute!` macro that works analogously for the Viper refute statement.
- Propose a way how to incorporate dead code checking into Prusti and implement it.
- Analogously do the same for unsoundness checking.
- Investigate how the dead code detection can interfere with unsoundness checking and propose a solution to make these two optional features coexist.

References

[1] Vytautas Astrauskas, Peter Müller, Federico Poli, and Alexander J Summers. Leveraging rust types for modular specification and verification. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):1–30, 2019.

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

[3] Jonas Maier. Towards verifying real-world rust programs. 2022.

[4] 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*, pages 41–62. Springer, 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. In *International Conference on Computer Aided Verification*, pages 367–379. Springer, 2021.