Verifying Rust Programs Using Floating-Point Numbers and Bitwise Operations

Bachelor Thesis Project Description

Lukas Friedlos
Supervised by Vytautas Astrauskas, Dr. Christoph Matheja, Prof. Dr. Peter Müller
Department of Computer Science
ETH Zürich
Zürich, Switzerland

I. Rust

Rust is a modern programming language suitable for systems programming with a heavy focus on safety and concurrency. It gives the programmers strong memory safety guarantees. To make programming in Rust even safer beyond the mechanism provided by the compiler, verifying the behavioral correctness of Rust programs is desirable.

II. Prusti

Prusti is a verifier that allows verifying the behavioral correctness of Rust programs [1]. Users can give specifications using annotations in the code. See Listing 1.

Prusti verifies the annotated Rust programs by translating them to the verification language of Viper [2], making use of Viper's verification infrastructure. Viper produces an SMT formula describing the constraints of the user given specifications and the program semantics, which is then passed to an external solver.

```rust
extern crate prusti_contracts;
use prusti_contracts::*;

#[pure] // side-effects free
#[requires(x >= 0 && y >= 0)]
#[ensures(result >= 1)]
fn foo(x: i32, y: i32) -> i32 {
    2 * x + y + 1
}
```

Listing 1: Example Rust program using integers with the annotations specifying that the function `foo` is pure, as well as a pre- and postcondition of the function. Other types include other integer types (`i*`, `u*`, `char`), `bool`, `ref`'s and composite types. As specifications one can also express loop invariants, predicates, absence of panics, or state specifications as fact declaring them `#[trusted]` [3].

Prusti currently supports all Rust integer types [4] and their associated operations. To handle a larger class of Rust programs, it would be beneficial to allow for the verification of operations on floating-points and bitwise operations. Such that Rust programs using types `f32`, `f64` and bitwise operations such as `&`, `|`, `^` or `<<` can be verified. Support for these types and operations has been added to Viper [5]. One *core goal* will be to give Prusti access to these features.

III. Floating-Points

Floating-Points are widely used by programmers. Rust's speed and native safety features make it a suitable language for many applications using FPs. Prusti should be able to verify such programs, especially since programs using FPs tend to be error-prone due to the idiosyncratic behavior of FP operations. See Listing 2 and 3.

```rust
1 fn foo() -> () {
2   let x: f64 = 0.4 + 0.2;
3   if x == 0.6 {
4     unreachable!();
5   }
6 }
```

Listing 2: The statement in line 4 is unreachable, since `0.4 + 0.2` evaluates to `0.6000000000000001` under addition of the IEEE 754 64-bit floating points used by Rust [6], [7].

Viper uses Z3 to solve SMT instances. Z3 adheres to the SMT-LIB standard description of IEEE 754 floating-points [8]. Support for this and other SMT-LIB types has been added to the Viper intermediate language [5]. A *core goal* of this thesis is to make these SMT-LIB types accessible to Prusti. This goal is twofold. First, it involves collecting information from the Rust compiler and encoding it into the Viper language. Second, it requires assessing how the resulting implementation handles various FP-specific behavior and devise methods for mitigating possible problems.

Xsat is another SMT solver, which doesn't directly reason about FP semantics, but instead solves an equivalent mathematical optimization problem [9], [10]. The cited paper shows this method achieved 100% consistency with SMT-solvers MathSat and Z3, as well as an avg. speedup of more than 700X on its limited benchmark (16 programs). An *extension goal* is to explore how Prusti or Viper could make use of Xsat to potentially exploit its speed.

```rust
fn bar(x: f64) -> f64 {
    if x < 1.0 {
        x
    } else if x >= 1.0 {
        -x
    } else {
        f64::NAN
    }
}
```

Listing 3: If the input `x` of function `bar` is close to `1.0` the representation error may change the branch outcome so that the mathematical form of the output is ambiguous [11]. In addition, the else branch is reachable with input `f64::NAN`.

Some errors, such as error propagation or the irregular branching around a threshold from Listing 3 can only be mitigated and some operations will always have executions with unstable behavior [11]. Heuristics have been devised for finding inputs triggering such errors [12],[13]. On the other hand, a new approach using dynamic analysis promises to ”detect more significant errors in real-world code [...] and achieve several orders of speedups over the state-of-the-art” [14]. An *extension goal* is to explore the feasibility and benefits of adding such a verification tool to Prusti/Viper.

Viper has a built-in type `Rational` for unbounded rational numbers. All FPs except `+inf`, `-inf` and `NaN` can be represented as a rational number. An *extension goal* is to give Prusti the ability to model FPs as rational numbers during the verification process.

IV. Bitwise Operations

Given the heavy focus on systems programming in Rust's design, a lot of Rust code is lower-level and uses bitwise operations for speed, or to interact with devices. Allowing for bitwise operations on bitvectors to be verified would be an important addition to Prusti. Viper allows for such verifications analogously to floating-points, using the SMT-LIB type for fixed-size bitvectors [5]. Implementing support for Rust programs using bitwise operations in Prusti is a *core goal* of this thesis.

In many programs, bitwise operations and operations on integers are used alongside each other. See Listing 4. Since these types of operations pertain to different theories, their interleaving poses questions about the approach on how to apply them. While all operations are supported by the bitvector theory, performance could be an issue compared to integers. To asses this, different approaches to applying the theories will be compared regarding efficiency and precision of analysis. Drawing from insights of this analysis, the thesis will describe a methodology for handling bitwise operations during the verification process as a *core goal*.

```rust
1  /// Calculates GCD using Stein's algorithm
2  fn gcd(&self, other: &Self) -> Self {
3      // ...
4      if m == 0 || n == 0 { return m | n; }
5      // find common factors of 2
6      let shift = (m | n).trailing_zeros();
7      // divide n and m by 2 until odd
8      m >>= m.trailing_zeros();
9      n >>= n.trailing_zeros();
10
11     while m != n {
12         if m > n {
13             m -= n;
14             m >>= m.trailing_zeros();
15         } else {
16             n -= m;
17             n >>= n.trailing_zeros();
18         }
19     }
20     m << shift
21 }
```

Listing 4: Implementation of `gcd` from the crate `num-integer` [15]. Note that in lines 13/14 and 16/17 an integer operation and a bitwise operation are applied alternatingly to `m` and `n`.

V. Core Goals

1) Collect a set of (real-world) Rust programs using floating-points and bitwise operations.
2) Devise a methodology for handling bitwise operations during the verification process.
3) Implement support for floating-points and bitwise operations in Prusti.
4) Apply the proposed features to verify properties of programs collected for 1).

VI. Extension Goals

5) Evaluate feasibility, benefits of possible approaches to make Xsat accessible to Prusti/Viper for FP verifications.
6) Evaluate feasibility, benefits of possible approaches to check for unstable behavior in FP programs.
7) Add manual proof support for non-linear arithmetic using floating-points to Prusti.
8) Allowing FPs in Rust to be modeled as rational numbers in Prusti.
9) Add support for the crate `bitflags` [16] and/or the crate `bitvec` [17] for programs using bitvectors of arbitrary length.

References

[1] V. Astrauskas, P. Müller, F. Poli, and A. J. Summers, “Leveraging Rust types for modular specification and verification,” in

Object-Oriented Programming Systems, Languages, and Applications (OOPSLA), vol. 3, no. OOPSLA. ACM, 2019, pp. 147:1–147:30.

[2] P. Müller, M. Schwerhoff, and A. J. Summers, "Viper: A verification infrastructure for permission-based reasoning," in *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, ser. LNCS, B. Jobstmann and K. R. M. Leino, Eds., vol. 9583. Springer-Verlag, 2016, pp. 41–62.

[3] P. D. Team. (2021) Prusti user guide. [Online]. Available: https://viperproject.github.io/prusti-dev/user-guide/

[4] R. D. Team. (2021) Rust data types. [Online]. Available: https://doc.rust-lang.org/book/ch03-02-data-types.html

[5] V. D. Team. (2020) Pull request 428: Backend support for smtlib types (particularly bitvectors and floats). [Online]. Available: https://github.com/viperproject/silver/pull/428

[6] R. D. Team. (2021) Primitive type f64 documentation. [Online]. Available: https://doc.rust-lang.org/std/primitive.f64.html

[7] "Ieee standard for binary floating-point arithmetic," *ANSI/IEEE Std 754-1985*, pp. 1–20, 1985.

[8] M. Brain, C. Tinelli, P. Ruemmer, and T. Wahl, "An automatable formal semantics for ieee-754 floating-point arithmetic," in *2015 IEEE 22nd Symposium on Computer Arithmetic*, 2015, pp. 160–167. [Online]. Available: http://smtlib.cs.uiowa.edu/papers/BTRW15.pdf

[9] Z. Fu and Z. Su, "Xsat: A fast floating-point satisfiability solver," in *Computer Aided Verification*, S. Chaudhuri and A. Farzan, Eds. Cham: Springer International Publishing, 2016, pp. 187–209.

[10] Z. Fu. (2019) Xsat github repository. [Online]. Available: https://github.com/zhoulaifu/xsat

[11] T. Bao and X. Zhang, "On-the-fly detection of instability problems in floating-point program execution," *SIGPLAN Not.*, vol. 48, no. 10, p. 817–832, Oct. 2013. [Online]. Available: https://www.cs.purdue.edu/homes/xyzhang/Comp/oopsla13.pdf

[12] W.-F. Chiang, G. Gopalakrishnan, Z. Rakamaric, and A. Solovyev, "Efficient search for inputs causing high floating-point errors," *SIGPLAN Not.*, vol. 49, no. 8, p. 43–52, Feb. 2014. [Online]. Available: https://doi.org/10.1145/2692916.2555265

[13] D. Zou, R. Wang, Y. Xiong, L. Zhang, Z. Su, and H. Mei, "A genetic algorithm for detecting significant floating-point inaccuracies," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1, 2015, pp. 529–539. [Online]. Available: https://damingz.github.io/papers/icse15.pdf

[14] D. Zou, M. Zeng, Y. Xiong, Z. Fu, L. Zhang, and Z. Su, "Detecting floating-point errors via atomic conditions," *Proc. ACM Program. Lang.*, vol. 4, no. POPL, Dec. 2019. [Online]. Available: https://doi.org/10.1145/3371128

[15] Various. (2021) `rust-num/num-integer` github repository. [Online]. Available: https://github.com/rust-num/num-integer/blob/master/src/lib.rs

[16] T. R. P. Developers. (2021) bitflags crate page. [Online]. Available: https://crates.io/crates/bitflags

[17] A. Payne. (2021) bitvec crate page. [Online]. Available: https://crates.io/crates/bitvec