Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Verifying Rust Programs Using Floating-Point Numbers and Bitwise Operations

Bachelor Thesis

Friedlos Lukas

September, 2021

Advisors: Vytautas Astrauskas, Dr. Christoph Matheja, Prof. Dr. Peter Müller
Department of Computer Science, ETH Zürich

Abstract

The verification of floating-point and bitvector operations is notoriously difficult, and inefficient at best. Nonetheless, both features are necessary in many applications. Hence, they should be supported by a modern verifier such as Prusti. Prusti targets the Rust programming language and is developed by the Programming Methodology Group at ETH Zurich. It allows its users to verify whether Rust programs adhere to user given specifications. Although Prusti is already quite rich in features and can verify a large set of programs, it does not support the use of floating-points or bitwise operations as of yet. As part of this thesis, support for these types and operations has been implemented for Prusti, increasing the set of programs supported by the verifier.

The thesis elaborates on the difficulty of said verification problems and shows example applications where verification does and does not succeed. It also describes a static analysis that helps determining the integer variables that need to be internally modeled as bitvectors in the verification process.

Contents

Chapter 1

Introduction

The use of floating-point numbers has proven to be error-prone over the many years since hardware floating-point units in processors have become widespread [14]. Subsequently, floating-point numbers have found common use among programmers for various applications. Many errors stem from the discrepancy between floating-point numbers and the real numbers they are trying to approximate. Listing 1, for example, shows a Rust function where a small rounding error in the addition of two floating-point numbers leads to an error when assuming that floating-point numbers behave like real numbers.

This 'unintuitive' behavior makes it desirable to verify programs using floating-point numbers in order to obtain reliable correctness guarantees. In the past, Pen and paper proofs turned out to be an insufficient solution, since they tend to be error prone themselves [5], they lack scalability and it requires great expertise to construct them. Reasoning about floating-points thus needs to be integrated in state-of-the-art verification tools.

Prusti is such a verification tool for the Rust programming language [1] by the Programming Methodology Group at ETH Zurich. Users can verify user-given specification, which are expressed on the level of the target language itself. This makes the tool very accessible. However, it currently

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

Listing 1: Rounding causes `0.2 + 0.4` to evaluate to `0.6000000000000001`, making the `if`-branch unreachable.

does not support the verification of floating-points and their operations. This thesis describes how the verification of programs using such operations was implemented in Prusti. It also shows practical limitations of verifying floating-points in general, and those specific to this implementation.

Another feature Prusti is currently lacking is the verification of bitwise operations (i.e. `&`, `|`, `^`, `<<`, `>>`). These operations act on integers in the Rust language (See Listing 2), but their verification requires the use of the theory of bitvectors. Prusti currently models integer types as mathematical (unbounded) integers. However, the corresponding theories of integer arithmetic do not support bitwise operations. While the theory of bitvectors does support these operations, it is significantly less efficient than the theory of integers. They share some of the inefficiencies of floating-point verification. Section 3.2 elaborates on these issues specific to the verification bitwise operations via bitvectors. Hence, it is desirable to keep the more efficient representation whenever possible. To this end, the thesis describes a static analysis which determines which integer-variables to treat as bitvectors, and which can remain integers for the verification. The implementation of said analysis and other changes made to Prusti in order to make bitwise operation verifiable are described in Section 5.3.

```
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      m >>= m.trailing_zeros();
8      n >>= n.trailing_zeros();
9      while m != n {
10         if m > n {
11             m -= n;
12             m >>= m.trailing_zeros();
13         } else {
14             n -= m;
15             n >>= n.trailing_zeros();
16         }
17     }
18     m << shift
19 }
```

Listing 2: Implementation of `gcd` from the Rust crate `num-integer`. Note that in lines 11/12 and 14/15 an integer operation and a bitwise operation are applied alternatingly to `m` and `n`.

The verification of both of these types of operations is an important part of a modern verification tool. Floating-points are in wide use and their verification is desirable, especially in safety-critical applications. And given Rust's focus on systems programming, where low-level operations are common and perform vital safety-critical functions, the ability to verify bitwise operations in a Rust program is arguably even more crucial.

Chapter 2 provides preliminary information about the programming languages, and verification frameworks relevant to this thesis. The difficulties associated with the verification of floating-point and bitwise operations are discussed in Chapter 3. Chapter 4 & 5 are concerned with the concrete implementation for each type of operation in Prusti.

Chapter 2

Background

This Chapter gives an overview of the programming languages, libraries, and verification frameworks this thesis is concerned with. Namely *Rust*, a modern systems programming language, and *Viper*, a verification language and infrastructure which can reason about a programs behavior. And *Prusti*, which serves as a bridge between the two and let's users verify Rust programs using Viper.

2.1 Rust

Rust is a modern programming language developed for systems programming with a heavy focus on safety and concurrency. It gives the programmers strong memory safety guarantees.

Many programming errors, which have plagued programs in older languages, can't occur in Rust. These errors include:

- Dereferencing a null pointer
- Not freeing memory after it's no longer used
- Double-freeing memory
- Accessing a dangling pointer

This is achieved by Rust's type system. It enforces a strong typing discipline which rules out the various memory safety issues mentioned above.

The development in computing towards concurrency has also been addressed in Rust by providing crucial thread-safety guarantees [21]. They include data race freedom and not acquiring the lock before accessing the memory it protects.

Rust allows its users to write both low-level and high-level code with all the conveniences of a modern programming language with zero-cost abstrac-

tions [20]. As such, Rust presents itself as a suitable programming language for both systems programming and high-performance applications, where the use of both bitwise operations and floating-points are common. Reasoning about them is crucial for the verification of real-world Rust programs. And it would allow the verification of the behavioral correctness of Rust programs beyond the guarantees provided by the compiler.

2.2 Viper

The Programming Methodology Group has developed a *Verification Infrastructure for Permission-based Reasoning* known as Viper [15]. It consists of its own intermediate verification language which serves as an abstraction to make the underlying verification tools accessible. The purpose of this infrastructure is to facilitate the development new verification tools. By providing a sound translation from programs in another programming language into the Viper language, one can leverage the existing verification infrastructure of Viper. A number of so-called *front-ends* for established programming languages (e.g. Go, Python, Rust, Java) have already been built and are still being developed. A recent addition to the Viper language is the ability to use floating-points and bitvectors [16].

2.3 Prusti

Prusti is a plug-in for the Rust compiler which allows the user to annotate their program with functional specifications, which it then attempts to verify [1]. Rust code and user given specifications are translated into Viper code, which is then verified using the Viper infrastructure, and the verification result is reported back to the user.

Since Rust's type system has very strict rules about ownership and aliasing, the guarantees provided by the Rust compiler can be leveraged and translated to Viper. This translation is done with the automated generation of a so-called 'core proof', concerned with the memory safety of the program [1]. Without this, these constraints would have be provided manually, since the core-proof is a crucial part of any proof about the correctness of a program which uses a heap and references.

Normally, the construction of such a 'core proof' is done manually, which is difficult and tedious, especially for non-experts. With the approach described above, Prusti can lift a heavy burden off the shoulders of the person who tries to verify a program. It allows its users to only concern themselves with higher-level functional properties of the program, for which they can write specifications on the level of Rust expressions. This greatly increases the accessibility to a verification tool for the Rust language.

2.3.1 The Verification Pipeline

Prusti takes a Rust file as file as input and first uses the Rust compiler to expands macros (functional specifications are given through macros, see Listing 3 line 4-6) and perform type checking.

From the Rust compiler, Prusti takes the MIR representation of the functions in the given file which ought to be verified. MIR is an intermediate representation used by the Rust compiler, where information about a program is more extensive and accessible.

The MIR is subsequently encoded into VIR (Viper Intermediate Representation), which is Prusti's own intermediate representation which mimics the Silver language for Viper. This additional IR is used to perform various analyses and optimizations specific to Prusti.

The VIR which results after optimizations is then translated to the Viper language, which can be passed to Viper for the verification.

As a last step, the Viper verifier is called on the Viper program and the result is reported back to the user.

2.3.2 Features and Usage

Here is an incomplete list of Prusti's features:

- Prusti can check for the absence of panics. If panics such as `panic!()` or `unreachable!()` are used in a program, Prusti will check whether these statements are reachable and report an error if they might be. See Listing 4 for an example.
- For assertions in a program of the form `assert!(...)`, Prusti tries to prove that the stated assertion always holds. It will report an error if it fails to do so. See Listing 5.
- Functions can be annotated with pre- and postcondition of the form `#[requires(...)]` and `#[ensures(...)]` respectively. The precondition is assumed to hold in the function body, while the postcondition is tried to be verified by Prusti to hold at the end of any execution of the function. See Listing 3 lines 5 and 6. If Prusti fails to verify, it is reported that the postcondition might not hold.
- Functions can be designated as pure functions, i.e. that they terminate and behave like mathematical functions. Pure functions can be designated by adding `#[pure]` before their declaration. Pure functions can then be used for expressing pre/post-conditions and assertions.

```
1  extern crate prusti_contracts;
2  use prusti_contracts::*;
3
4  #[pure] // side-effects free
5  #[requires(x >= 0 && y >= 0)]
6  #[ensures(result >= 1)]
7  fn foo(x: i32, y: i32) -> i32 {
8      2 * x + y + 1
9  }
```

Listing 3: Example Rust program with functional specifications to be verified by Prusti, given as macros. The user given specifications are in lines 4-6. The function is designated as side-effect free in line 4. An assumed precondition is specified in line 5, and a postcondition to be asserted is found in line 6.

```
1  fn foo(b: bool) {
2      if b { unreachable!() }
3  }
```

Listing 4: Program where a panic is reachable. The panic statement in line 2 can obviously reached if `b` is `true`. Prusti will report that the statement "might be reachable".

```
1  fn foo(x: i64) {
2      let abs = if x < 0 {
3          -x
4      } else{
5          x
6      };
7      assert!(abs >= 0)
8      return abs
9  }
```

Listing 5: Rust program with an assertion to be verified by Prusti. The function `foo` returns the absolute value of a signed 64-bit integer. The assertion in line 7 is verified to hold for any execution. Note that the same property could be verified by adding the condition to be asserted as a postcondition above the function declaration.

Chapter 3

Verification Challenges

The verification of programs using floating-points or bitwise operations is notoriously difficult [14, 2], and many obstacles to fast verification have not yet been overcome [13, 14]. Though some of their problems are shared and have the same origin, there are also difficulties which are specific to floating-points and bitwise operations respectively.

3.1 Floating-Points

Guarantees for programs using floating-points are generally rather low. This is at least in part due to the difficulty of making precise analyses, stemming from rounding and exceptions [5]. Related to this is the lack of algebraic structure to be exploited for rewriting and simplifying expressions [13]. This, among other reasons, causes the formulas expressing the constraints to be solved by an SMT-solver to be very large, and hence their verification to be slow. This section expands on these issues pertaining to floating-point verification.

Floating-point Errors Floating-points serve as an approximation to real numbers. But their limited size makes them unable to accurately represent most of them. Consequently, floating-point operations are not closed, meaning [4] :

1. Results may exceed the value of the largest floating-point value causing an *overflow*;
2. Results may be smaller than the smallest non-zero floating-point value causing an *underflow*; and
3. Results may lie between two adjacent floating-point values causing it to be *imprecise* and requiring rounding

Such errors can accumulate in the course of an execution leading to *error-propagation*. These errors are inherent to floating-points and can only be mitigated [3]. Though heuristics have been devised for finding specific inputs and input ranges which lead to a program execution where these errors occur [23, 8]. Most approaches use static analysis based on interval arithmetic or affine arithmetic [23]. Newer techniques for finding such inputs using symbolic execution promise significant speedups compared to the established methods [24]. On a brighter note, for many programs it may be the case, that only a small set of inputs cause large errors of this nature [3].

Lack of Structure to Exploit An inherent theoretic obstacle to fast verification of floating-point operations is the fact that many algebraic properties of other number systems do not hold for floating-points [13]. E.g. addition and multiplication are not associative. So `(a + b) + c == a + (b + c)` does not hold in general for floating-points `a`, `b`, `c`. Therefore much fewer expressions can be rewritten and simplified, compared to (for example) rational numbers. See Listing 6 for an example where a desired simplification is not allowed.

```
1  extern crate prusti_contracts;
2  use prusti_contracts::*;
3
4  #[requires(x != 0.)]
5  fn foo(x:f64) {
6      assert!(x * (1./x) == 1.); // Does not hold
7  }
8  fn main() {}
```

Listing 6: Although the assertion in line 6 holds for some instantiations of the variable `x`, it does not hold in general.

A major source for such issues arising when working with floating-points are the *special values NaN*, *Inf*, *-Inf*, as well as the non-uniqueness of 0 (i.e. $+0/-0$). Although they allow for a more convenient usage of floating-points, since less interrupts are thrown in the execution of operations, they can easily lead to program crashes at another point of a programs execution, or allow for unintuitive execution paths (See Listing 7).

NaN also breaks the reflexivity of comparison operators. Since any comparison to *NaN* evaluates to *False*, so does `f64::NAN == f64::NAN`, single handedly making equality non-reflexive. See Section 4.2.1 for a mitigation technique.

```
1 fn foo(x: f64) -> f64 {
2   if x < 1.0 {
3       x
4   } else if x >= 1.0 {
5       -x
6   } else {
7       unreachable!() // actually reachable!
8   }
9 }
```

Listing 7: One might think that the cases `< 1.0` and `>= 1.0` cover all cases. But note that the `else`-branch in line 6 is in fact reachable. If the function `foo` is called with `f64::NAN`, the branch is taken and the program would crash.

Large Formulas Most practical limitations stem from the fact, that the SMT solver has to resort to *bit-blasting* when dealing with floating-points. Bit-blasting means that floating-point numbers are considered as bitvectors, and all possible assignments of the bits are considered. This results in huge formulas for the solver to resolve, resulting in long runtimes for the verification, even for relatively simple programs.

Another reason for large formulas is difficulty of rewriting and simplifying expressions involving floating-points described in the previous paragraph.

Conclusion There is little mathematical structure which can be exploited when reasoning about operations on floating-point [7], and a prover is most often forced to resort to *bitblasting* to solve constraints on a program using floating-points.

3.2 Bitwise Operations

Bitwise operations (see Table 3.1) act on *fixed-sized* bitvectors. Integers of both the signed and unsigned type are such bitvectors of fixed size. When this thesis refers to bitvectors, it means *fixed-sized* bitvectors, particularly of the size of standard integers (8, 16, 32, 64 and 128-Bit).

The verification of integer arithmetic is much easier for solvers to handle, since constraints can more easily be rewritten and simplified, resulting in shorter formulas for an SMT-solver. With bitvectors, however, the possibilities for rewriting is limited, and doing so very often does not reduce the problem size sufficiently [9, 14]. This is mainly because the algebraic properties of integers lend more opportunity for simplification and reorganization of constraints than the structure of the bitvector semantics.

Table 3.1: Bitwise Operations

Operation	Rust Operator	Description
BitAnd	`&`	Bitwise And
BitOr	`|`	Bitwise Or
BitXor	`^`	Bitwise Xor
Shl	`<<`	Bitwise Left-Shift
Shr	`>>`	Bitwise Right-Shift

In addition, more research and development has been made for solving integer constraints than constraints on bitvectors, and the mentioned disadvantage of bitvectors may at be least partially mitigated by new techniques and insights from research into the latter, resulting in more powerful applications [9].

Since integer constraints can be handled more efficiently than bitvector constraints, but both may be needed to formulate constraints on integers (e.g. both a standard arithmetic operation and a bitwise operation are applied to te same integer variable) the question arises as to *how to use the two theories in combination?* The goal in resolving this issue to resort to bitvector minimally in order to reduce the size of the constraint-formulas.

We can therefore say, there are two main difficulties when trying to verify programs using bitwise operations:

1. The mingling of the theory for integers and the theory for bitvectors.
2. Lack of exploitable structure for rewriting constraints resulting in large formulas.

Note though, that the first issue is directly linked to the second one, for reasons mentioned above.

```
1      ...
2  let x: u32 = 1 << 5;
3  let y: u32 = x + 1;
4      ...
```

Listing 8: Code snippet where the variable `x` comes into contact with both a bitwise operation and an arithmetic integer operation.

Bitvectors and Integers: Two Colliding Theories Since we have some operations (`+`, `-`, ...) which we would like to verify in the theory for integers and some operations (`&`, `|`, ...) which we have to verify in the theory for bitvectors, and these two sets of operations act on the same objects (integers), the question arises how we handle situations where the two collide,

see Listing 8. Section 5.2 presents a methodology for handling this issue and elaborates on it further.

Large Formulas The main issue with bitvector verification is that the formulas to solve grow exponentially with the number of variables. An SMT-solver often would take an unreasonable amount of time to solve a formula, or chokes prematurely if the given formula is too large. The only way to combat this is with rewriting and simplification techniques of the formulas themselves.

Modern solvers employ a range of rewriting and simplification techniques to reduce the size of the formula to solve. And they make a significant effort to achieve this [22].

Newer techniques, together in combination with traditional ones, can achieve magnitudal speedups for quantified bitvector formulas [22].

Another approach is to evaluate the constraints lazily and breaking them up in smaller parts. Such a technique not only achieved a speedup, but managed to solve some constraints where the eager rewriting techniques and wholesale evaluation failed [9].

Techniques developed by research are often developed for one specific SMT-solver, and take a long time to find their way into other deployed SMT-solvers, if at all.

Chapter 4

Verifying Floating-Point Operations

Floating-point numbers are widely used by programmers in various applications. They serve as an accessible approximation to the real numbers. Apart from their use in scientific applications and in data science, they also find uses in safety-critical systems, e.g. in aerospace or medical devices. Rust is generally a suitable programming language for safety-critical system, given its design as a modern systems programming language. Its speed, portability and native safety features concerning memory errors provide a good foundation for the next generation of applications of the aforementioned kind. Programs using floating-points tend to be error prone due to the idiosyncrasies of the floating-point semantics, and the unintuitive behavior of floating-point operations arising from them (see Section 3.1 for examples). Prusti should be able to verify programs which use floating-points and thereby giving the programmers additional guarantees about the functional behavior of a program in addition to the safety guarantees provided by the Rust compiler. This chapter describes how the verification of floating-points and their operations is implemented in Prusti. It also explores the limits of the described implementation's capabilities.

4.1 Implementation

4.1.1 Overview

The goal of the implementation is to allow the verification of Rust Programs which use Rust's native floating-points `f32` and `f64` and some of their associated operations.

Rust floating-point types follow the IEEE 754 standard for floating-points [10] and it uses the 'round to nearest, ties to even' rounding mode. Viper uses Z3 as its SMT solver (Section 1.3) to check the whether the specifications hold for a given program. Z3 supports the SMTLib standard [6] for IEEE

floating-points [19]. With the pull request #428 [16] to the Silver language repository (back-end of Viper) this feature of Z3 has been made accessible in the Viper verification infrastructure.

The implementation described in this thesis makes this new functionality of the Viper back-end accessible to Prusti. To produce a valid Viper program corresponding to a given Rust program with Prusti specification syntax, Prusti takes the MIR form of the program from the Rust compiler, produces its own Viper Intermediate Representation (VIR), before converting the VIR to a Viper program. See 2.3.1 for details. In order to encode floating-points in Viper, the implementation interjects itself in these two stages. Specifically when creating the VIR from the MIR, and when converting the VIR-AST to Viper. Maybe include a graphic outlining what is done at each conversion.

4.1.2 FloatFactory from the Silver Language

The change to the Silver language mentioned above added AST nodes representing various floating-point types and their associated functions [16]. This makes the SMTLib theory for floating-points accessible in the Viper back-end. In the course of writing the implementation for Prusti minor bugs in the aforementioned pull request where found and remedied by Vytautas Astrauskas [17, 18].

Most important to our usage of this new functionality is Silver's `FloatFactory` class, which let's us create SMTLib-Floats, and operate on them. See Table 4.1 for an overview of the floating-point operations provided by the `FloatFactory`.

A `FloatFactory(mant: Int, exp: Int, roundingMode: RoundingMode)` is created by providing it with the mantissa and exponent size of a float type, as well as the rounding mode we wish to use. In Listing 9 you can see an example of a specific `FloatFactory` being created.

```
1 FloatFactory(52 , 12, RoundingMode.RNE)
```

Listing 9: Creation of a `FloatFactory` for 64-bit floating-points using the "Round to nearest, ties to even" rounding mode. Note that 64-bit floating-points have 52 bits for the mantissa, 11 bits for the exponent, and 1 bit for the sign. So the sign bit is to be included in the number of bits of the exponent.

Used Features	Comments
`from_bv`	Create floating-point from a bitvector. Used in the creation of a floating-points constant.
`neg`	Invert a floating-point. Is called whenever a unary minus operation is applied to a floating-point variable or constant.
`isNaN`	Checks whether a floating-point represents a *NaN*-value. Calls to `is_nan` are intercepted and converted to calls of this `FloatFactory`-function.
`add`, `sub`, `mul`, `div`	Arithmetic operations defined by the IEEE standard on two floating-points. Rust uses of `+`, `-`, `*` and `/` are converted to calls of these functions.
`min`, `max`	Min/Max of two floating-points. Used analogously to `isNaN`.
`eq`, `leq`, `geq`, `lt`, `gt`	Comparison operators on floating-points. They are called analogously to the arithmetic operations.
Unused Features	**Comments**
`abs`	Absolute value of a floating-point. Calls could be intercepted analogously to `min/max`.
`isZero`, `isInfinite`, `isNegative`, `isPositive`	Currently unused, but they are accessible Prusti. May be used for rewriting equivalent expressions, or to intercept calls to equivalent built-in functions.

Table 4.1: Overview of the functions on floating-points provided by the `FloatFactory`.

4.1.3 MIR to VIR

In this step, the aim is to take the information from the MIR provided by the Rust compiler, and encode it into the *Viper Intermediate Representation* VIR. The variables of a Rust program with type `f32` or `f64` are encoded in VIR as floats as well. To this end, a new type for floating-points has been added to VIR. See Listing 10.

Additionally a new kind of constant expression has been added to VIR in order to represent the floating-point constants which appear in the MIR. See Listing 11 how constant floating-points are encoded in VIR. There are constant types for both of the two sizes of Rust floating-points (`f32` & `f64`), where the actual value of the constant is encoded as a `u32` or `u64` respectivley. The concrete value will be converted to a bitvector first, before we can convert it to a float again in Viper at a later stage. That's why the representation as a bit-pattern was chosen already at this stage.

The built-in functions on floating-points `is_nan`, `min` and `max` are manually intercepted and encoded as VIR-expression which are translated to calls to

the corresponding functions of the `FloatFactory` in the next stage where VIR is translated to Viper.

```
pub enum Type {
    ... // Previously existing types
    Float(FloatSize),
}

pub enum FloatSize {
    F32,
    F64,
}
```

Listing 10: VIR float type. The Rust type `f64` corresponds to the VIR type `Type::Float(FloatSize::F64)`.

```
pub enum Const {
    ... // Previously existing kinds of constant expressions
    Float(FloatConst),
}

pub enum FloatConst {
    FloatConst32(u32),
    FloatConst64(u64),
}
```

Listing 11: VIR Const expression for floating-points, where `u32`/`u64` hold the bit-pattern of the float it represents. E.g. `1.0: f32` is represented as `Const::Float(FloatConst32(1065353216))`.

4.1.4 VIR to Viper

A unary or binary operation on floating-points is converted to their respective function call of an appropriate `FloatFactory` (depending on the size of the floats).

Constant floating-points are first converted to a bitvector. The BVFactory (See Section Section 5.3) function `from_nat` is called on the `u32`/`u64` value of the `FloatConst` (See Listing 11). This is followed by a call of the `FloatFactory` function `from_bv` on the bitvector.

4.2 Verifying Programs

In general, the verification of programs using floating-points is difficult and slow. The reasons for which are both practical and theoretical (Section 2.3.1). This chapter deals with how these difficulties can and cannot be mitigated in this particular implementation.

4.2.1 Verification in Practice

In general, programs which use `f32` were much easier and faster to verify than programs using `f64`. While this is unsurprising, it is noteworthy that for some programs, it was not possible to verify it with `f64`'s, but verified easily after changing types to `f32`.

Especially programs using division on floating-points are near impossible to verify (in reasonable time). Even a very simple program as the one in Listing 12 did not finish verification after an hour. Interestingly enough, when changing the precondition to `1. < x && x < 1.1` the verification terminates successfully after a few seconds. Faster termination with strict inequalities has been observed in other programs as well. Whether this is due to the specific implementation in Prusti, or inherent to the SMT-solver was not explored in the course of this thesis.

```
1  #[requires(1. <= x && x <= 1.1)]
2  #[ensures(result >= 0.)]
3  fn foo(x:f32) -> f32 {
4
5      return 1. / x
6
7  }
```

Listing 12: Simple Rust function using only one floating-point division.

Section 2.3.1 mentions some theoretic obstacles to verifying programs with floating-points. In particular the non-reflexivity of equality comparisons of floating-points. In practice, one seems to be able to restore the reflexive property of the equality comparisons `==`, `<=` and `>=` easily by assuming floats not to be *NaN*. See Listing 13 below. The function `refl` asserts the reflexivity of said comparison operations, while `is_not_nan` is used to formulate the mentioned assumption as a precondition. Note that the function `is_not_nan(x)` is functionally equivalent to `!x.is_nan()`. But the built-in function `core::f32::<impl f32>::is_nan` cannot be used in specifications since its implementation is impure. In general, calls to `is_nan` are possible, and they are translated to calls of the `isNAN` function of the `FloatFactory`.

```
1  extern crate prusti_contracts;
2  use prusti_contracts::*;
3
4  #[requires(is_not_nan(a))]
5  fn refl(a:f32) {
6      assert!(a == a);
7      assert!(a <= a);
8      assert!(a >= a);
9  }
10
11 #[pure]
12 fn is_not_nan(a:f32) -> bool {
13     f32::NEG_INFINITY <= a && a <= f32::INFINITY
14 }
```

Listing 13: This program verifies the reflexivity of comparison operators of 32-bit floating-points under the assumption that they are not *NaN*.

4.2.2 Verifying Real-World Programs

Angle Difference So far the only publicly available Rust code that uses floating-points successfully verified with our extension of Prusti is a solution given for an algorithm which should calculate the angle difference of two bearings given in degrees. It was posted on the rosettacode web page [1]. Provided with two angles given as floating-points, the program should return the difference in angle in a range from -180 to 180. See Listing 15.

In order to reach some successful verification, the original code had to be modified, and an assumption about the input angles had to be added. See Listing 15. First, in line 12, the original code from the solution is left as a comment. This code had to be replaced with line 13 since there is no support for the floating-point modulo operator provided by the `FloatFactory`. It is not part of the SMT-Lib theory for floating-points, since it is also not part of the IEEE standard for floating-points. The definition of the Rust modulo on floating-points can be seen in Listing 14 [2]. So if one were to be able to mimic the truncation, the Rust modulus for floating-points could be implemented for Prusti. To remedy this, a precondition is added to the function `angle_difference` in line 9 which ensures that the difference between the two given angles is already reduced. The same precondition also ensures that no special floating-point values are passed as arguments. Considering these modifications, the informative value of this verification about the behavior of the original program is questionable.

[1] https://www.rosettacode.org/wiki/Angle_difference_between_two_bearings
[2] Source corde: https://doc.rust-lang.org/src/core/ops/arith.rs.html#583

```
x % y = x - (x / y).trunc() * y
```

Listing 14: Definition of the Rust modulo on floating-points.

```
1  extern crate prusti_contracts;
2  use prusti_contracts::*;
3
4  #[pure]
5  fn is_reduced(angle: f32) -> bool {
6      -360. < angle && angle < 360.
7  }
8
9  #[requires(is_reduced(bearing2 - bearing1))]
10 #[ensures(-180. <= result && result <= 180.)]
11 pub fn angle_difference(bearing1: f32, bearing2: f32) -> f32 {
12     // let diff = (bearing2 - bearing1) % 360.0; original code
13     let diff = bearing2 - bearing1;
14     if diff < -180.0 {
15         360.0 + diff
16     } else if diff > 180.0 {
17         -360.0 + diff
18     } else {
19         diff
20     }
21 }
```

Listing 15: Rust implementation of "Angle difference between two bearings" from rosettacode.

Additionally, the program from Listing 15 is one of the programs which verify successfully with `f32`'s, but if the type annotations are replaced with `f64` the verification fails. It fails with the error "the post-condition might not hold" referring to line 10.

Uniform Distribution In the Prusti Crate `probability` [3] one can find a number of suitable functions to test the implementation on. It contains various utilities for calculating with probabilities as the name suggests. The structs and functions are fairly simple, and mathematical properties they ought to adhere to are well known. It also finds public use with an average of about 50 downloads per day.

The part of this crate which was given the most effort to verify was its implementation of the distribution function for continuous uniform distributions

[3] https://crates.io/crates/probability

[4]. You can see the function and the relevant struct in Listing 16. Trying to verify whether the return values of the function are in fact in the codomain of a distribution function, which is $(0, 1)$. The verification could never be run to completion in reasonable time. Changing the size of the floating-points to 32-bit, or weakening the post-condition and using strict inequalities (e.g. `-0.1 < result && result < 1.1`), or both, made no discernible impact on the runtime of the verification.

```
1  struct Uniform {
2      a:f64,
3      b:f64,
4  }
5
6  #[requires(d.a < d.b)]
7  #[ensures(0. <= result && result <= 1.)]
8  fn distribution(d:Uniform, x: f64) -> f64 {
9      if x <= d.a {
10         0.0
11     } else if x >= d.b {
12         1.0
13     } else {
14         (x - d.a) / (d.b - d.a)
15     }
16 }
```

Listing 16: Implementation of continuous uniform distributions from the `probability` crate.

4.3 Benchmarks

Function Arguments In order to determine how the number of arguments to a function impacts the runtime of verification, a set of tests where run using the benchmarking tool of Prusti. The benchmarked functions all used three 32-bit floating-point numbers, which where either arguments to the function with the assumption that they are larger or equal to `0.0`, or random floating-point constants between `0.0` and `100.0`. The function returns the sum of all these floating-points and verifies, that their sum is larger or equal to `0.0`. See Listing 17.

[4] https://github.com/stainless-steel/probability/blob/master/src/distribution/uniform.rs

```
1  #[ensures(result >= 0.0)]
2  fn foo_0() -> f32 {
3      let _0: f32 = 85.70717120125501;
4      let _1: f32 = 44.66243410754219;
5      let _2: f32 = 16.569445079123557;
6      let _acc0: f32 = _0 + _1;
7      let _acc1: f32 = _acc0 + _2;
8      return _acc1
9  }
10
11 ...
12
13 #[requires(_0 >= 0.0)]
14 #[requires(_1 >= 0.0)]
15 #[ensures(result >= 0.0)]
16 fn foo_2(_0: f32, _1: f32) -> f32 {
17     let _2: f32 = 9.540560509896267;
18     let _acc0: f32 = _0 + _1;
19     let _acc1: f32 = _acc0 + _2;
20     return _acc1
21 }
22
23 ...
```

Listing 17: Two functions of the benchmark described above.

Results:

No. of Arguments	**Avg[s]**	**Var**
0	0.423	0.00099
1	1.133	0.00006
2	14.957	84.9956
3	14.2199	0.18493

Note the large variance in runtime for the function with only two arguments passed. This result corresponds to the function in line 16 from Listing 17. Initial suspicion that this is caused by a bug in the benchmarking tool could be dismissed by running the test directly, measuring the verification time via the shell. For 40 runs in this setup the verification time varied between 5 and 260 seconds. For functions with only one argument and three arguments, the verification times were consistent. We currently have no explanation for this behavior, and its causes could be investigated in future work.

The benchmarking performed in the course of this thesis is limited, and further benchmarking would be helpful in finding which kinds of programs can be verified efficiently.

Conclusion Although the implementation works, and self-made programs (see the tests in `prusti-test/verify/[pass,fail]/floats/*`) are correctly verifiable in reasonable time, the verification of most actual programs in use, which surpass a trivial level of complexity do not seem to be able to be verified in a reasonable amount of time.

Chapter 5

Verifying Bitwise Operations

5.1 Preliminaries

Fixed-sized integers of both the signed and unsigned type are bitvectors. So Prusti already supports some bitvectors, as well as some operations on them which can be verfied using the SMT-theory for integers. However, for some of the operations which can be performed on integers, a different theory is needed to handle their verification. In particular, these are the *bitwise operations* (Table 3.1). See also Table 5.1 for which theory supports which operations.

Table 5.1: Operations Supported by Theory [11, 12]

Operation	Integer Theory	Bitvector Theory
Add	✓	✓
Sub	✓	✓
Mul	✓	✓
Div	✓	✓
Neg	✓	✓
Mod	✓	✗
Abs	✓	✗
Geq/Leq	✓	✓
Gt/Lt	✓	✓
Not	✗	✓
BitAnd	✗	✓
BitOr	✗	✓
BitXor	✗	✓
Shl	✗	✓
Shr	✗	✓

There exists an established SMT-theory for fixed-sized bitvectors [11], which will be used to verify bitwise operations in the implementation decribed in this chapter. So when this chapter mentions bitvectors, it always refers to bitvectors of *fixed size*.

Since it is much faster to verify programs with integers (See Section 3.2), we would like to treat bitvectors as integers whenever possible without resorting to the SMT-theory for fixed-sized bitvectors.

The following section describes a naive methodology which determines all variables and constants which are directly or indirectly affected by bitwise operations, in order to treat them as bitvectors during verification. Furthermore, it explains the changes which had to be made to Prusti to make bitvector verification possible, including the concrete implementation of the static analysis in Prusti.

5.2 Towards an Efficient Use of the Bitvector Theory

5.2.1 Static Analysis for Determining Bitvector-Taint

The motivation for this analysis is the desire to use bitvectors sparingly for efficiency reasons. So instead of treating all integers as bitvectors in a program as soon as a bitwise operation appears, this approach allows integers to remain integers if they do not come into contact with a bitwise operation either directly or indirectly.

So the goal is to determine which integers are used in a bitwise operation or come into contact with such an operation transitively.

In the following description we make the following simplifications: All variables are references to be dereferenced by a field of a certain type. There is only one size of integer and each function of a program is treated separately. The following procedure is applied to each function of a program:

We hold a set $\mathcal{T}$ of variables which are *tainted* by bitwise operations.

1. Iterate through the statements of the function.
   - If a statement contains a bitwise operation, add the operands to $\mathcal{T}$.
   - If a tainted variable is connected through a binary operation or an assignment to other variables, add them to $\mathcal{T}$.
2. If $|\mathcal{T}|$ has increased since the beginning of step 1, repeat step 1. Else return $\mathcal{T}$.
3. Pass through the statements, changing all dereferences of the variables in $\mathcal{T}$.

Example In this example, the variable declarations are left out. All variables are references to a certain type. And the values they refer to are accessed via the correct field beginning with `val_type`. So if `_x` is a variable referring to an integer, the integer is accessed with `_x.val_int`.

```
1 method foo() {
2     _0.val_int := 1
3     _1.val_int := 2
4
5     _2.val_int := _0.val_int << 4
6     _3.val_int := _1.val_int + 1
7     _4.val_int := _2.val_int + 1
8 }
```

Changes to the set of tainted expressions by iteration:

1. `_0` is added to $\mathcal{T}$ in line 5, since it's an operand of the bitwise shift operation.
2. `_2` is added in line 5 since the right side contains taint. And in line 7 the variable `_4` get added to $\mathcal{T}$ due to the taint in the right-hand side of the assignment.
3. No additions to the tainted variables. $\mathcal{T} = \{$`_0`, `_2`, `_4` $\}$.

After changing the dereferences, the program looks like this:

```
1 method foo() {
2     _0.val_bv := 1
3     _1.val_int := 2
4
5     _2.val_bv := _0.val_bv << 4
6     _3.val_int := _1.val_int + 1
7     _4.val_bv := _2.val_bv + 1
8 }
```

5.2.2 Deficiencies and Other Possible Approaches

The methodology describe above is obviously a naive. It is sound, since it moves everything directly or indirectly affected by the bitvector theory in its realm. But given the fact that bitvector verification is slow, a method which minimizes its use is more desirable.

Another (naive) solution would be to simply convert integers to bitvectors before a bitwise operation, and convert it to an integer after the operation. Informally, the idea is to remain in the realm of the integers with brief excursion to the bitvectors theory for the bitwise operations. See Listing 18 for an example. This naive approach could be improved by omitting 'unnecessary'

conversions (e.g. if bitwise operations are applied subsequently, one could omit the two conversions between the operations). To determine the absolutely necessary conversions a static analysis is needed to determine where conversions are to be placed.

This approach may not be desirable either, since these conversions are costly and it is questionable if it would result in better runtimes.

```
1       ...
2   let x: u8 = 13;
3   let y: u8 = x & 2;
4       ...
```

Pseudo-code containing the needed conversions:

```
1   ...
2   x = 13
3   y = toInt(toBV8(x) & toBV8(2))
4   ...
```

Listing 18: Example of using conversions to handle bitwise operations and integers clashing

Speculation: More efficient approaches are to be found somewhere between these two methods. The goal of such an approach should probably be to limit the spread of taint, by finding well placed conversion which hinder the wide spread of taint to such an extent that it would reduce the runtime of the verification. See Listing 19. To find this balance a considerable amount of empirical analysis on runtimes would be required.

```
1   let v: u8 = 42;
2   let w: u8 = 1;
3   let x = v + w;
4   let y = x - 1;
5   let z = v << 2;
6   assert!(z & 1 == 1);
```

Listing 19: Example of how bitvector-taint can affect a whole program even though it is highly local. Line 5 adds `v` and `z`. Then in line 3 `v` affects `w` and `x`. And line 4 finally adds `y` to the tainted variables. So all variables are marked to be bitvectors, but only `z` 'needs' to be for the BitAnd operation in line 6. So instead of tainting all variables, two conversions would suffice. Note also that `v << 2` could be replaced with `v * 2 * 2`.

Used Features	**Comments**
`from_int`, `from_nat`	Create bitvector from a signed/unsigned integer. Used to convert constant integers to bitvectors.
`neg`	Unary minus.
`add`, `sub`, `mul`, `udiv`	Arithmetic operations on bitvectors.
`and`, `or`, `xor`, `not`	Bitwise operations on bitvectors.
`shl`, `lshr`, `ashr`	Shift operators, where logical right shift (`lshr`) is used on Rust's `u*` integers and arithmetic right shift (`ashr`) `i*` integers.
Unused Features	**Comments**
`to_int`, `to_nat`	Conversion to a signed/unsigned integer.
`nand`, `nor`, `xnor`	Bitwise operations with no equivalent operator in Rust.

Table 5.2: Overview of the functions on bitvectors provided by the `BVFactory`

5.3 Implementation

Apart from the implementation of the static analysis in Section 5.3.3, this implementation's general approach is analogous to the implementation for floating-point numbers in Section 4.1.

5.3.1 The Silver BVFactory

The change to the Silver language mentioned in the implementation for floating-point functionality also added AST nodes representing SMT-Lib fixed-sized bitvectors and their associated functions [16]. Such bitvectors are created using the `BVFactory` class. It is also through this class that on access the functions operating on bitvectors. A `BVFactory(size: Int)` is created by providing it with the size of the bitvector. The functions on bitvectors provided by the factory can be found in Table 5.2.

5.3.2 Changes to VIR

A new type has been added to the VIR-AST for bitvectors of the sizes of the built-in integers of Rust. See Listing 20 below. Note that expressions of this type are only created as part of the later described static analysis. This is in contrast to the floating-points, which are encoded directly from the MIR.

```
pub enum Type {
    ... // Previously existing types
    Bitvector(BVSize),
}

pub enum BVSize {
    BV8,
    BV16,
    BV32,
    BV64,
    BV128,
}
```

Listing 20: VIR bitvector type. The enum variants of `BVSize` correspond to the sizes of native integers in Rust.

I order to represent constant bitvectors, a new type of constant expression has been added as well. See Listing 21 below. The bit-pattern of a constant bitvector is stored as an unsigned integer of the same size.

```
pub enum Const {
    ... // Previously existing kinds of constant expressions
    Bitvector(BVConst),
}

pub enum BVConst {
    BV8(u8),
    BV16(u16),
    BV32(u32),
    BV64(u64),
    BV128(u128),
}
```

Listing 21: VIR Const expression for bitvecotrs, where the unsigned integer in the `BVConst` enum hold the bit-pattern of the bitvector it represents. The bitvector constants in the final viper program are created by a call to `from_nat` on these unsigned integers.

Unary and binary operations on bitvectors are then turned into calls of the respective function from an appropriately sized `BVFactory` when the VIR is translated to the Viper language. Here the size of the bitvectors is needed to create the factory. This information is taken from the type.

5.3.3 Static Analysis Implementation

The broad approach to implementing the analysis goes as follows:

1. Run analysis on each method after optimizations.
2. Modify method according to the result of the analysis.
3. Store relevant information in the `Encoder`-struct.
4. Add new predicates
5. Add new fields

All code relevant to this implementation is located in `bitvector_analysis.rs`. Its main component is the `BVAnalysis`-struct and its associated functions in `impl BVAnalysis`, as well as other helper functions. See the `BVAnalysis`-struct fields and their purpose int the code in Listing 22.

```
1     pub struct BVAnalysis {
2         // Method to be analyzed
3         method: CfgMethod,
4         // Map from all tainted fields and variables to their replacements
5         replacements: HashMap<Expr, Expr>,
6         // New fields which are used in the replacements
7         pub new_fields: HashSet<Field>,
8         // New predicates for predicates called on tainted variables
9         pub new_predicates: HashSet<String>,
10 }
```

Listing 22: Declaration of the `BVAnalysis`-struct.

The static analysis is performed after optimizations have been performed on a method in `procedure_encoder.rs`. This is done by creating a new `BVAnalysis`-struct and calling its `get_new_method` function with the method to be analyzed. This functions fills the fields of the `BVAnalysis`-struct and returns the method with the appropriate replacements performed. The analysis itself can be performed by calling the struct's `run` method. There is a call to this function inside `get_new_method` as well.

The analysis follows the description of the analysis in the previous section and takes the following form in Rust code:

```
let mut continue_analysis = true;
while continue_analysis {
    let pre_size = self.replacements.len();
    for blk in &basic_blocks {
        for stmt in &blk.stmts {
            self.check_stmt(&stmt);
        }
    }
    continue_analysis = pre_size < self.replacements.len();
}
```

The function `check_stmt` matches the given statement and checks the statements substatements and/or expressions with `check_stmt` and `check_expr` respectively. The only statement-kind given special treatment are *assign*-statements. If the right-hand side of the assignment is tainted, the left-hand side and its replacement are added to the `replacements`-map.

The function `check_expr` matches the given expression, and if it is a bitwise binary operation, is adds the subexpressions and their replacements to `replacements`.

5.3.4 Determining Replacements of Tainted Expressions

The replacement type is determined by the reference type of the tainted local variable. Say the analysis determines, that `_x.val_int` is tainted. And `_x` has type `TypedRef(u32)`, the replacement is `_x.val_bv32`, where `_x` now has type `TypedRef(bv32)`. In the same turn `_x` with its new type is also added as a replacement for the old `_x`. And `bv32` with type `Bitvector(BVSize::BV32)` is added to `new_fields`.

For composite types such as tuples or structs, constructing correct replacements gets more involved. With tuples, for example, the string describing the reference type of the variable has to be correctly altered. Say, we have a variable `_x` of the following type:

```
Ref(tuple2$u8$u8)
```

And we found that the field use `_x.tuple_0` is tainted. We need to get the index of the tainted tuple value from the field name, in this case `0`. So the type of the first element of the tuple has to be changed, such that the the replacement variable has the following type:

```
Ref(tuple2$bv8$u8)
```

For structs the same has to be done, but instead of an index we have to work with the name of the struct field. This part of the implementation was not built to completion by the end of the thesis. The main difficulty lies in

the correct string manipulation and especially for more nested structs and tuples, the implementation crashes regularly.

5.3.5 Limitations

The implementation is ripe with errors and regularily fails in testing. Especially the use of composite types such as structs and enums often results in errors, because of the difficulty of extracting and propagating the necessary information from nested fields to construct all the correct replacements. The thesis failed to engineer a complete and error-proof way of doing this. This and the lack of sophistication in the methodology for handling the combined use of the integer and bitvector theory suggests a new approach and implementation is desirable in the long term.

Chapter 6

Conclusion

Verifying programs using floating-points and bitwise operations is hard, and the verification process of both is lacking in efficiency.

Concerning floating-points operations, the implementation written as part of this thesis allows Prusti to verify programs using native floating-points. In practice however, it proves to be very difficult to arrive at meaningful guarantees about the behavior of said floating-points. This is due to the idiosyncrasies of floating-points, causing their verification to be very inefficient.

The support for bitwise operations is still lacking and not ready for deployment. The static analysis described in this thesis can be used for the verification of programs using bitwise operations. But future work may find better ways of handling the combined use of bitvectors and integers in verification. But even with a more efficient verification technique, an implementation may face the same limitations as encountered with floating-points.

The current verification tools still require research and improvement to allow for *practically efficient* verification of floating-points and bitwise operations. An empiric study into which kinds of operations/programs are efficiently verifiable and which are practically impossible to verify at the current state of verification tools may also be the subject of future work.

Bibliography

[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)*, volume 3, pages 147:1–147:30. ACM, 2019. doi:10.1145/3360573.

[2] Peter Backeman, Philipp Rümmer, and Aleksandar Zeljić. Interpolating bit-vector formulas using uninterpreted predicates and presburger arithmetic. *Formal Methods in System Design*, May 2021. URL: https://doi.org/10.1007/s10703-021-00372-6, doi:10.1007/s10703-021-00372-6.

[3] Tao Bao and Xiangyu Zhang. On-the-fly detection of instability problems in floating-point program execution. *SIGPLAN Not.*, 48(10):817832, October 2013. URL: https://www.cs.purdue.edu/homes/xyzhang/Comp/oopsla13.pdf, doi:10.1145/2544173.2509526.

[4] Earl T. Barr, Thanh Vo, Vu Le, and Zhendong Su. Automatic detection of floating-point exceptions. 48(1):549560, January 2013. URL: https://people.inf.ethz.ch/suz/theses/TV-thesis.pdf, doi:10.1145/2480359.2429133.

[5] Sylvie Boldo and Jean-Christophe Filliatre. Formal verification of floating-point programs. In *18th IEEE Symposium on Computer Arithmetic (ARITH '07)*, pages 187–194, 2007. URL: https://www.lri.fr/~filliatr/ftp/publis/caduceus-floats.pdf, doi:10.1109/ARITH.2007.20.

[6] 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*, pages 160–167, 2015. URL: http://smtlib.cs.uiowa.edu/papers/BTRW15.pdf, doi:10.1109/ARITH.2015.26.

[7] Martin Brain, Cesare Tinelli, Philipp Ruemmer, and Thomas Wahl. An automatable formal semantics for ieee-754 floating-point arithmetic. In *2015 IEEE 22nd Symposium on Computer Arithmetic*, pages 160–167, 2015. URL: http://smtlib.cs.uiowa.edu/papers/BTRW14.pdf, doi:10.1109/ARITH.2015.26.

[8] Wei-Fan Chiang, Ganesh Gopalakrishnan, Zvonimir Rakamaric, and Alexey Solovyev. Efficient search for inputs causing high floating-point errors. *SIGPLAN Not.*, 49(8):4352, February 2014. URL: https://doi.org/10.1145/2692916.2555265, doi:10.1145/2692916.2555265.

[9] Liana Hadarean, Kshitij Bansal, Dejan Jovanović, Clark Barrett, and Cesare Tinelli. A tale of two solvers: Eager and lazy approaches to bit-vectors. In Armin Biere and Roderick Bloem, editors, *Computer Aided Verification*, pages 680–695, Cham, 2014. Springer International Publishing.

[10] IEEE. Ieee standard for floating-point arithmetic. *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, pages 1–84, 2019. doi:10.1109/IEEESTD.2019.8766229.

[11] SMT-Lib Initiative. Smt-lib theory of fixed sized bitvectors, 2021. URL: https://smtlib.cs.uiowa.edu/theories-FixedSizeBitVectors.shtml.

[12] SMT-Lib Initiative. Smt-lib theory of integers, 2021. URL: https://smtlib.cs.uiowa.edu/theories-Ints.shtml.

[13] Wonyeol Lee, Rahul Sharma, and Alex Aiken. Verifying bit-manipulations of floating-point. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, page 7084, New York, NY, USA, 2016. Association for Computing Machinery. URL: https://theory.stanford.edu/~aiken/publications/papers/pldi16b.pdf, doi:10.1145/2908080.2908107.

[14] David Monniaux. The pitfalls of verifying floating-point computations. *ACM Trans. Program. Lang. Syst.*, 30(3), May 2008. URL: https://hal.archives-ouvertes.fr/hal-00128124/file/floating-point-article.pdf, doi:10.1145/1353445.1353446.

[15] Peter Müller, Malte Schwerhoff, and Alexander J. Summers. Viper: A verification infrastructure for permission-based reasoning. In Barbara Jobstmann and K. Rustan M. Leino, editors, *Verification, Model Checking, and Abstract Interpretation*, pages 41–62, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.

[16] Viper Development Team. Pull request 428: Backend support for smtlib types (particularly bitvectors and floats), 2020. URL: https://github.com/viperproject/silver/pull/428.

[17] Viper Development Team. Pull request 527: Fix encoding of fp.min and fp.max and add a test, 2021. URL: https://github.com/viperproject/silver/pull/537.

[18] Viper Development Team. Pull request 537: Fix bvlshr name and add a test, 2021. URL: https://github.com/viperproject/silver/pull/537.

[19] Z3 Development Team. Z3. floatingpoint module documentation, 2021. URL: https://z3prover.github.io/api/html/ml/Z3.FloatingPoint.html.

[20] Aaron Turon. Abstraction without overhead: traits in rust, 2015. URL: https://blog.rust-lang.org/2015/05/11/traits.html.

[21] Aaron Turon. Fearless concurrency with rust, 2015. URL: https://blog.rust-lang.org/2015/04/10/Fearless-Concurrency.html.

[22] Christoph M. Wintersteiger, Youssef Hamadi, and Leonardo de Moura. Efficiently solving quantified bit-vector formulas. In *Formal Methods in Computer Aided Design*, pages 239–246, 2010.

[23] 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*, volume 1, pages 529–539, 2015. URL: https://damingz.github.io/papers/icse15.pdf, doi:10.1109/ICSE.2015.70.

[24] Daming Zou, Muhan Zeng, Yingfei Xiong, Zhoulai Fu, Lu Zhang, and Zhendong Su. Detecting floating-point errors via atomic conditions. *Proc. ACM Program. Lang.*, 4(POPL), December 2019. URL: https://doi.org/10.1145/3371128, doi:10.1145/3371128.

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

Verifying Rust Programs Using Floating-Point and Bitwise Operations

Authored by (in block letters):

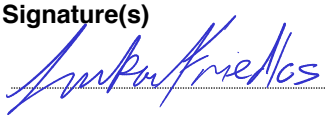
For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
Friedlos	Lukas

With my signature I confirm that

- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	**Signature(s)**
Zurich, 16.08.21	

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.