Reasoning about Complexities in a Rust Verifier

Master's Thesis Project Description

Lowis Engel
Supervised by Dr. Christoph Matheja and Aurel Bílý

ETH Zürich

March 2021

1 Introduction

The Rust programming language [6] has attracted a lot of attention in recent years [9, 8] due to its focus on strong safety guarantees while keeping the capabilities of a low-level systems programming language. Its type system statically prevents common memory bugs such as data races, dangling pointers and undesirable aliasing side effects.

Nevertheless, memory safety alone does not make a program functionally correct. Empirical methods like testing can be used to increase the confidence in a program's correctness. However, if one wants to fully prove the absence of errors, a static verifier should be used. Such verifiers take a program together with a formal specification and attempt to prove that the specification is fulfilled.

The Prusti project built one such verifier for Rust [1]. By leveraging the guarantees provided by the Rust compiler, it is able to hide complicated logic from the user that is otherwise needed for reasoning about heap-based programs. After the type checking by the Rust compiler, Prusti obtains the functions to verify in the Mid-level Intermediate Representation (MIR) and translates them together with the specification into the intermediate representation of the Viper infrastructure [7]. Viper's verifier is then used as a back-end to perform the actual verification.

Currently, the Prusti verifier is only able to verify functional requirements and the absence of errors like overflows or panics. However, in many cases programs should not only be functionally correct, but also efficient. This is especially true for a systems language like Rust targeting low-level applications. It is often easy to write a simple correct program that is not very efficient, but correctly optimizing it while keeping functional correctness is hard. Therefore, we would like to verify non-functional properties such as bounds on the (asymptotic) runtime or space usage at the same time as the functional requirements. Being able to verify both simultaneously is also beneficial because some complexity proofs might rely on knowledge from the functional correctness proof.

Related Work

To enable formal reasoning about the resource usage of programs, Robert Atkey formalized amortized complexity analyses in separation logic [2]. He transferred the notion of separating access permissions of heap locations to the concept of 'consumable resources' which enables 'consuming' them only once. This approach was then brought into practice in [3] by verifying explicit bounds for the worst-case amortized time complexity of a Union-Find data structure. They use so-called *time credits* to represent the permission to perform a single operation in their cost model.

Building on this previous work, Armaël Guéneau proposed in his PhD thesis [4] to abstract the cost of a function by using asymptotic bounds. This required him to formalize the 'Big-O notation' and the composition of such bounds to keep modularity. Using abstract bounds improves modularity, since small changes in the program do not require specifications to change. This also simplifies the specifications to an often sufficient level of abstraction. Moreover, working with asymptotic bounds makes the verification more robust against the concrete choice of a cost model and therefore more independent from platform details. This enables Guéneau to use a very simple, but elegant, cost model where only entering a loop or function body has a cost, which is 1.

Example

We would like to use the results from this previous work to verify (asymptotic) complexity specifications as shown in Example 1 with Prusti. The use case in this example is the verification of typical claims in library specifications. For instance, Rust's standard library states that its `Vec` implementation 'guarantee[s] O(1) amortized `push`'[1]. The example shows a simplified version of this `Vec` implementation for which we formalize the claim with a possible syntax for specifications.

```rust
#[stored_credits(runtime(O(2*self.len - self.capacity))]
impl<T> Vector<T> {
    #[requires(runtime(O(self.len))]     // for copying to new location
    #[ensures(self.capacity == total_new_cap)]
    #[ensures(self.len == old(self.len))]
    pub fn reserve(&mut self, total_new_cap: usize) {...}

    #[requires(amortized(runtime(O(1)))]
    pub fn push(&mut self, value: T) {
        if self.len == self.capacity {
            if self.capacity == 0 {
                self.reserve(1);
            }
            else {
                self.reserve(self.capacity * 2);
            }
        }
        self.buffer[self.len] = value;
        self.len += 1;  // costs O(1) credits due to change of stored_credits
    }
}
```

Example 1: Amortized analysis of a resizing vector push

Intuitively, one can prove the amortized runtime by assuming that we store some constant amount of time credits per vector element added by a `push` call. These credits are then used when the capacity is reached, to pay for copying the lower-half (with potentially no credits left) as well as the elements from the upper-half themselves to the newly allocated location. This credit view is similar to the 'banker's view' described in [10]. Having control over time credit *store* and *pay* operations may be needed in some use cases. Especially for data structures like linked lists without an explicit length, stating resource properties at each element and therefore relating them to the data structure's shape as in [2] might also be useful.

In this example it is arguably the easiest for static verification to use the so-called 'potential' or 'physicist's' approach to amortized analysis [10]. This means, that we define a potential function which maps the state of the data structure to an amount of time

[1] `https://doc.rust-lang.org/std/vec/struct.Vec.html`

credits currently stored for later use. The amortized cost of an operation then consists of the actual cost of the operation plus the change of potential caused by it. If the potential grows, we will pay for later, costlier operations. These operations can then make use of the stored credits by decreasing the potential.

In the specification syntax of the example we define the potential function by using the `stored_credits` keyword in line 1. We want to be flexible in terms of the consumable resource to analyze to allow future extensions, for example to memory space complexity. Therefore, we use the keyword `runtime` in the annotations to define which resource we want to reason about. As mentioned before, we want to use the abstraction of the 'Big-O notation' in most use cases, but sometimes concrete bounds might also be necessary and are also easier to start with. That is why we introduce the `O` to signal the use of asymptotic complexities.

For the specification of function complexities we use the standard Prusti `requires`- and `ensures`-clauses for pre- and postconditions. Using them in preconditions might seem more natural if one thinks in terms of time credits to pay. Whereas, complexities in postconditions can be interpreted as credit debt after the call and allow the usage of result values in the annotation. For the `push` function we want to prove an amortized complexity. Therefore, we use the `amortized` keyword in the specification to enable the usage of the `stored_credits`. For `reserve` on the other hand we only give a non-amortized worst-case complexity, which might be useful if one wants to reason about the strict worst-case complexity of a program without amortization.

To prove the specification of `push`, an automatic verifier would step through the MIR statements of the function and construct a verification condition for its correctness with respect to the annotations. For the cost inference during this construction it would use the costs given for each operation in a cost model. As mentioned before, the change of potential will be added to the operation's cost when performing an amortized analysis.

The `reserve` call would be analyzed modularly by only using its specification and not the implementation. Although the annotations only give a non-amortized runtime in this case, we can infer the amortized runtime easily by adding the change of potential. Here we usually call `reserve` with the doubled capacity. By the postconditions we can infer that `self.capacity` will be set to the new capacity and `self.len` will be left unchanged. This leads to a decrease of potential by the old `self.capacity`. Since we know that `self.len` is equal to this capacity by line 10, the amortized complexity of the call is `O(self.len) - O(self.len)`, which is equal to `O(1)`. It can be easily seen that the edge case, when `self.capacity` is zero, also costs at most `O(1)`, since `self.len` is zero. The remaining parts of the function can also be performed in constant time, since we only change the potential by a constant amount on line 20.

Following the approach in [4], the verifier would probably first construct a concrete cost expression for the code. To keep the verification modular this cost expression would contain placeholders for the costs of function calls. Only in the end, we would prove that the cost expression satisfies the user-defined asymptotic bound by using the (asymptotic) bounds provided in the specification of the called functions as constraints.

2 Core Goals

The main goal of this thesis is to design and implement an extension of the Prusti verifier to enable modular verification of asymptotic runtime bounds for programs written in Rust. This task can be subdivided into the following core goals with a rough estimate of the required time in weeks on the right-hand side (in total about 16 of 26 weeks), which includes time needed for the implementation:

- **Collecting Examples:** Find interesting examples of Rust code to be verified. This includes examples exposing complexity bugs (i.e. correct programs that take longer than necessary and expected), but also correct examples that might not be simple to analyze. These can then be used to guide the designing phase and for the evaluation of the implementation. (2)
- **Cost Model:** Define a runtime cost model for Rust's MIR in terms of time credits. This model should be generic to allow verification with respect to different cost models and in particular also allow reasoning about the consumption of other resources such as memory (see Extension Goals). (2)
- **Specification Syntax:** Define syntax to reason about cost functions and (time) credits. As we have seen in Example 1, this mainly includes annotations to define complexity bounds asymptotically or concretely, which should be flexible in terms of different resources. But it may also include additional annotations which may be needed to guide the cost inference process as in [4]. In particular, loops will require annotations similar to a variant if the maximal number of iterations cannot directly be inferred. (1)
- **Viper Encoding:** Design an encoding of the cost model and specifications into Viper. This includes constructing a (concrete) cost expression for the MIR code, which can then be proven by Viper's verifier to satisfy the time credit bound given in the specification. (4)
- **Proving Asymptotic Bounds:** Encode rules into Viper to prove asymptotic upper bounds. This includes formalizing domination properties between functions and simplification steps for complexity bounds. In the current state of Viper this is probably only possible for uni-variate linear bounds. But it may be possible to achieve first results for more complex functions, such as polynomial, logarithmic or multivariate bounds (see Extension Goals). (3)
- **Amortized Analysis:** Make it possible to store and extract time credits to and from Rust data structures to enable amortized cost analyses. As mentioned, this can be done using the potential method as in Example 1, but more flexible solutions are needed in some cases. Try to minimize the manual annotations needed for this. (2)
- **Evaluation:** Evaluate the verification, for example by re-verifying bounds established by existing projects or modeling proofs of textbook algorithms. Use the examples collected before. A possibility for evaluation might also be to perform termination proofs by proving bounds on the runtime. (2)

3 Extension Goals

The remaining time of about 10 weeks will be used for writing the report, finishing delayed core goals and for implementing some of the following extensions:

- **Lower Bounds:** Extend the verifier to be able prove lower bounds on time complexity.
- **Space Complexity:** Add a cost model for memory consumption and verify space complexity bounds.

- **Complexity Inference:** Automatically infer (simple) asymptotic complexity bounds from the MIR code without a user providing a candidate bound.
- **More Complex Asymptotic Bounds:** Enable proof support for (more complicated) polynomial, logarithmic or multivariate bounds. Using the approach from Hoffmann et al. [5] to translate super-linear functions into linear constraints of base functions, which can be solved by a LP solver, could be useful. Another approach could be to require more manual annotations to enable non-linear inferences in Viper.
- **Timing Side Channels:** Prove that running times are independent of cryptographic secrets to show absence of timing side channels for security applications. For this approach more precise cost models are probably needed.

References

[1] V. Astrauskas et al. "Leveraging Rust Types for Modular Specification and Verification". In: *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA).* Vol. 3. OOPSLA. ACM, 2019, 147:1–147:30. DOI: `10.1145/3360573`.

[2] Robert Atkey. "Amortised Resource Analysis with Separation Logic". In: *Logical Methods in Computer Science* 7.2 (June 2011). Ed. by Andrew Gordon. DOI: `10.2168/lmcs-7(2:17)2011`.

[3] A. Charguéraud and F. Pottier. "Verifying the correctness and amortized complexity of a union-find implementation in separation logic with time credits". In: *Journal of Automated Reasoning* 62.3 (2019), pp. 331–365.

[4] Armaël Guéneau. "Mechanized Verification of the Correctness and Asymptotic Complexity of Programs". PhD thesis. Université de Paris, 2019.

[5] J. Hoffmann and M. Hofmann. "Amortized Resource Analysis with Polynomial Potential". In: *ESOP*. 2010, pp. 287–306.

[6] N. D. Matsakis and F. S. Klock. "The Rust Language". In: *Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology.* HILT '14. New York, NY, USA: Association for Computing Machinery, 2014, 103–104. ISBN: 9781450332170. DOI: `10.1145/2663171.2663188`.

[7] P. Müller, M. Schwerhoff, and A. J. Summers. "Viper: A Verification Infrastructure for Permission-Based Reasoning". In: *Verification, Model Checking, and Abstract Interpretation (VMCAI).* Ed. by B. Jobstmann and K. R. M. Leino. Vol. 9583. LNCS. Springer-Verlag, 2016, pp. 41–62.

[8] Ryan Levick. *Why Rust for safe systems programming.* Microsoft Blog Post. 2019. URL: `https://msrc-blog.microsoft.com/2019/07/22/why-rust-for-safe-systems-programming/` (visited on 02/23/2021).

[9] *Stack Overflow Developer Survey 2020 - Most Loved.* 2020. URL: `https://insights.stackoverflow.com/survey/2020#technology-most-loved-dreaded-and-wanted-languages-loved` (visited on 02/23/2021).

[10] Robert Endre Tarjan. "Amortized computational complexity". In: *SIAM Journal on Algebraic Discrete Methods* 6.2 (1985), pp. 306–318.