Verifying Competitive-Programming Programs in a Rust Verifier

Master's Thesis Project Description
Department of Computer Science, ETH Zurich, Switzerland
Department of Computer Science, TUM, Germany

Ahmed Rayan
Supervisors: Vytautas Astrauskas, Prof. Dr. Peter Müller, Prof. Dr. Susanne Albers

Start Date: October 1, 2020

Introduction

A big challenge in creating tasks for programming competitions such as ICPC and IOI is ensuring that the created master solutions are correct. The task authors almost always provide an (informal) argument why a specific solution is correct as shown in the appendix. However, the correctness of the implementation is checked only by testing and peer-review. Using these techniques to ensure the correctness of the master solution doesn't guarantee that it is fully correct. On the other hand, human error can be eliminated by providing an automated way to verify the correctness of the implemented master solution.

Generally, the verification of program correctness is notoriously difficult. Moreover, this task for competitive programs is even harder because they are typically written in C and C++ languages whose type-systems provide almost no guarantees.

Prusti[1] is a verification tool for Rust programs that tries to use the strong Rust type system to significantly simplify the verification. It allows developers to write contracts and annotations in the source code, which are then automatically checked to prove the functional correctness of the program. Internally, Prusti translates Rust programs to the Viper[2] intermediate verification language.

Since programs written in Rust have similar performance characteristics as the ones written in C and C++, Rust could be used as an alternative language to C and C++ in the programming competitions for writing master solutions. Having the option to write master solutions in Rust enables the use of Prusti for verifying these solutions.

In competitive programming, there are clear classes of tasks based on what patterns can be used for solving them. It seems that these patterns also correspond to the vocabulary used by

programmers in this field talking to each other. For example, saying that a problem will be solved using a minimum range segment tree gives the programmer an indication of the main structure of the solution program without even looking at the problem. But, two questions are arising here and we will try to find an answer for them while working on the project. The first question is whether these solving patterns can indicate guiding the verification of the solution as it gives the main outline for the solution itself. The second question is whether Rust vocabulary matches that of C and C++, which might change the verification of these algorithms using Prusti. For example, tree representation in C/C++ is usually done by a parent array where the value in each cell represents the index of the parent for the vertex with that index. However, the natural representation for a tree in Rust is done using a Rust struct with the subtrees.

The main goal of this project is to find strategies to verify solutions implemented in Rust for competitive programming problems.

In the following subsection, an example illustrating the verification for a Rust program to a competitive programming problem is presented followed by another subsection to show the main approach taken to achieve the goal of the project.

Example

The Fibonacci problem is one of the famous problems in the competitive programming community. It is considered a simple example of a problem that is solved using the Dynamic Programming (DP) technique. In this section, we are explaining how to verify the DP Rust solution for the Fibonacci problem.

Firstly, a recursive function that solves the problem directly by stating the base cases and calling the recurrence directly will be implemented as shown below.

```
#[pure]
fn fib_req(n: usize) -> i32 {
 if n <= 1 { 1 }
 else {
     fib_req(n-1) + fib_req(n-2)
 }
}
```

Now, it is time to implement our bottom-up DP solution in another function. The ultimate goal here is to ensure that the results from both functions are equal to each other.

```
#[ensures = "result == fib_req(n)"]
fn fib(n: usize) -> i32 {
 if n <= 1 {
       1
```

```
    } else {
        let mut fib = VecWrapperI32::new();
        fib.push(1);
        fib.push(1);
        let mut i = 2usize;
        #[invariant = "i >= 2"]
        #[invariant = "fib.len() == i"]
        #[invariant = "forall k: usize :: (k < i && k >= 0) ==> fib.lookup(k) == fib_req(k)"]
        while i <= n {
            let cur = fib.lookup(i - 1) + fib.lookup(i - 2);
            fib.push(cur);
            i += 1;
        }
        fib.lookup(n)
    }
}
```

Prusti translates this into Viper intermediate language and it manages to successfully verify the equality between the results of the two functions.

The main idea we used here is to add the equality of the sofar calculated values in the DP table and the recursive function result in the invariant of the while loop filling up the table. The key question here is whether this idea could be used also for other DP programs and what are the reasons if this is not the case.

Approach

To achieve the main goal of this project which is finding a strategy to verify the solution for some competitive programming problem based on what problem class it belongs to, the following steps will be the main guideline to follow:

1. A set of topics is to be selected. Each of these topics represents one pattern or technique for solving tasks in the competitive programming community. The aim is to find a verification pattern for each of these topics. Mainly, the book **Competitive Programming 3**[3] will be used to choose that set of topics as it provides an organized explanation for most of the common topics in the field.
2. For each of the selected topics, which also represents a problem class, a set of problems will be selected to verify their solutions. Competitive Programming 3 provides a large collection of problems for each topic contained in it. These problems also exist on uva online judge[4], which gives a way to submit solutions and make sure they are correct.
3. Translating the solution for each of the selected problems into Rust is to be performed. During the translation, we should pay attention to how natural the solution implementation in Rust is. This will help later to check how this aspect affects the simplicity of the verification for that specific problem.
4. Prusti will be used to verify the implemented solution for each problem pointing out the relation between how natural the implementation is and how easy the verification is. We need to check if It is easier to verify algorithm implementation that can be represented

naturally in Rust and how much easier it is. It is common to evaluate the verification effort by comparing the number of annotation lines to the lines of the original program.
5. For each topic, a general strategy for verifying its solutions will be derived from the verified solutions of the selected problems belonging to this topic.

There are a couple of risks to working on this project:

- Prusti is still a young tool and missing many features. we may get stuck and be unable to move on with the verification of the chosen problems due to the lack of support from Prusti. In such a case, changing the direction of the thesis project into choosing a set of features and supporting them in Prusti is possible.
- As verification is hard and not very straightforward, some of the chosen competitive programming topics might be unsuitable for verification using Prusti. This will leave us with the option to change these topics into more suitable ones.

Core Goals

The main core goals of this project are to try finding strategies for verifying the solutions to the problems under the topics of **Greedy**, **Dynamic Programming,** and **DP on Trees**. This will be done by following the explained approach for each of these topics.

Greedy

An algorithm is said to be greedy if it makes the locally optimal choice at each step with the hope of eventually reaching the globally optimal solution. Usually, the greedy solution is short and runs efficiently. For the problem to have a greedy solution, it must have the following two properties:

- **It has optimal substructures**. An optimal solution to the problem contains optimal solutions to the sub-problems.
- **It has the greedy property.** If we make a choice that seems like the best at the moment and proceed to solve the remaining subproblem, we reach the optimal solution. We will never have to reconsider our previous choices.

An example of a greedy problem will be the **Frog Jumping** problem explained in the appendix. We have a simple greedy algorithm for routing the frog home by jumping as forward as possible at each step. Here, the two previously explained properties hold as it has optimal substructure and the greedy property is choosing to jump to the farthest lilypad within the frog range.

An idea to verify the optimality of this algorithm is to assume any valid sequence of jumps from the start to the end and try to prove the sequence constructed using our greedy solution is at least as good as any assumed sequence.

Dynamic Programming (DP)

Another goal of the project is to use Dynamic Programming as one of the topics we aim to find a strategy for verifying the solutions to its problems. DP is primarily used to solve optimization problems and counting problems. For a problem to be solvable using the DP technique, It must exhibit two properties:

- **This problem has optimal sub-structures** which means the solution for the sub-problem is part of the solution of the original problem.
- **This problem has overlapping sub-problems.**

For this part of the project, we will use different problems having DP solutions. However, we will focus on problems that work on linear data structures or mathematical recurrences.

DP on Trees

Moving on from verifying DP solutions for problems with linear data structures into more advanced problems that work with trees is the main aim of this core goal. In this part of the project, we will examine how the choice of the way to represent the trees in Rust will affect the complexity of verifying the DP solution on such trees.

Extension Goals

As some extension goals for this project, There are two main paths to follow. The first one is to explore more advanced topics like segment trees and DAGs following the same approach used for the topics in the core goals. An alternative path is formalizing the informal Proof.

Exploring More Topics

Segment Tree

A segment tree is a data structure that is used mainly to solve dynamic range queries over a linear array efficiently in logarithmic time. It is a way to arrange data into a binary tree with several ways to implement it. One of the implementations of a segment tree is to use a 1-based compact array where each cell will be responsible for holding information about a specific continuous segment in the original array. Another way of implementing it is to use the Rust tree struct. The second way is the natural way to represent binary trees in Rust which is expected to make the verification easier for segment tree problems. A comparison between the verification for both representations will be done to check how much the choice of representation affects the complexity of the verification.

Directed Acyclic Graph (DAG)

Verifying solutions for graph problems is considered quite hard due to the lack of ordering of the vertices in the graph as graphs usually contain cycles. However, A DAG is a special kind of

graph that is directed and acyclic. For this kind of graph, we can find a linear ordering for the vertices which is called Topological Sort. Theoretically, having this ordering for vertices should make the verification for problems solutions that use DAGs more doable than the ones that use arbitrary graphs.

Normally, DAGs are represented in the implementation in the same way as any other graphs. So, the problem of ensuring the DAG property for the graph and using this as a part of the verification for the whole solution for the main problem is challenging and requires investigation.

Formalizing Informal Proofs

The previous goals aim to prove the correctness of the implementation following an informal proof. However, this extension goal would aim to formalize the informal proof in Prusti. To achieve that, we will formalize the problems by translating the textual form into a mathematical description for the inputs and describe the characteristics of the solution formally and mathematically. Then, we will try to adjust the verification of the implemented solution to meet the formally described inputs and desired output.

Schedule

1. Greedy: 3 weeks
2. Dynamic Programming (DP): 3 weeks
3. DP on Trees: 3 weeks
4. Extension goals: 6 weeks
5. Writing: 4 weeks

Appendix

In this appendix, we provide an example of a competitive programming task which is the **Frog Jumping** problem.[5]

The frog begins at position 0 in the river. Its goal is to get to position n. There are lilypads at various positions. There is always a lilypad at position 0 and position n. The frog can jump at most r units at a time. The goal is to find the path the frog should take to minimize jumps, assuming a solution exists.

We have a simple algorithm for routing the frog home by jumping as forward as possible at each step. Usually, when such a task is faced in the competitive programming community, we don't prove the solution formally. An informal argument is provided instead. Here, if the frog is at position k and it has the option to jump forward to one of the positions i and j , where $i > j$, then it is better to jump to j as every move the frog can do from the position i forward, it can

also do it from the position j. So, there is no advantage of the shorter jump and it is always better to jump as far as possible.

References

1. Astrauskas, V., Müller, P., Poli, F. & Summers, A. J. Leveraging rust types for modular specification and verification. *Proceedings of the ACM on Programming Languages* vol. 3 1–30 (2019).
2. Müller, P., Schwerhoff, M. & Summers, A. J. Viper: A Verification Infrastructure for Permission-Based Reasoning. *Lecture Notes in Computer Science* 41–62 (2016) doi:10.1007/978-3-662-49122-5_2.
3. Halim, S. & Halim, F. *Competitive Programming 3: The New Lower Bound of Programming Contests*. (2013).
4. Online Judge - Home. https://onlinejudge.org/.
5. CS161: Design and Analysis of Algorithms. http://web.stanford.edu/class/archive/cs/cs161/cs161.1138/.