Developing IDE Support for a Rust Verifier

Bachelor's Thesis Project Description

Julian Dunskus
Supervised by Federico Poli & Vytautas Astrauskas
under Prof. Dr. Peter Müller
Department of Computer Science, ETH Zurich
Zurich, Switzerland

February 2020

1 Introduction

Prusti [1] is a Rust [2] verifier developed by the Programming Methodology Group at ETH Zurich. It is based on the Viper [3] verification infrastructure: It translates input to a Viper AST and verifies that using Viper, then associates the results with the original Rust source. It can verify `assert!(...)` assertions in code as well as pre-/postconditions and invariants on functions (specified via Rust's attribute system).

Prusti Assistant is a plugin for Microsoft Visual Studio Code [4] that allows users to verify their code using Prusti without using the command-line interface. The main focus of this project is improving Prusti Assistant's integration with VS Code as well as its execution times.

2 Motivation

Currently, every time a file is verified, Prusti Assistant needs to launch Prusti, which in turn launches Viper, which takes around 5 seconds just to launch. Ideally, Prusti and/or Viper would already be running in server mode, removing those startup times from the verification process and thus speeding it up significantly—especially for smaller files, where the startup time is a much bigger factor than the actual verification time. This server could also be run remotely, reducing the load on the user's local machine and freeing up its resources.

In addition to this, the current IDE integration is rather basic in its functionality and the user experience has room for improvement. Especially when compared to other plugins like the Viper IDE [5], there are many helpful features it currently lacks, like a progress bar or a summary of the verification status in status bar. Further, a great hindrance for potential new users of Prusti Assistant is the need to perform manual installation steps, including installing a specific nightly build of Rust.

3 Approach

3.1 Core Goals

1. **Prusti Server Mode**
   In order to let Prusti run as a server, it needs to communicate with Viper. Viper can accept either a Viper source code file or an AST, of which Prusti uses the latter because it allows us to attach information about the relations to the original Rust source code. Viper already has a server mode, which however only accepts source code, so we have two options:

   (a) Extend Viper's existing server mode, adding a serialization format for the Viper AST, so we can serialize the AST in Prusti and deserialize it in the Viper server. This would also be useful for other verifiers who need to use an AST rather than source code.

   (b) Write a new server in Rust that manages its own instance of Viper, starting the verification tasks directly with the ability to provide ASTs as input.

2. **Improve Prusti IDE Integration:**

   (1) Remove manual steps from Prusti installation
   Prusti Assistant's automatic installation process (which currently handles installing Prusti itself as well as Viper) will be expanded to include installing the required Rust build via `rustup`.

   (2) Allow users to choose between stable and nightly builds of Prusti
   Since Prusti Assistant manages its installation of Prusti, an option can be added to its settings in VS Code to choose whether to use stable or nightly builds. This would also be useful functionality to generalize as a Node.js [6] module, for potential reuse by the Viper IDE mentioned earlier.

   (3) Status bar improvements
   VS Code's status bar is a handy space for information about the verification process, e.g. a progress bar for how far along it is or a quick overview of the verification results for the current file.

3.2 Extension Goals

1. **Enable users to add contracts to existing functions**
   In order to improve the general usefulness of Prusti, it would be very helpful to be able to verify code that calls into external (library) functions. Figuring out a way to specify contracts about those functions would go a long way towards making real-world Rust projects easier to verify.

2. **Optimize the plugin's performance using a profiler**
   In addition to the server mode, another way to further reduce the time it takes to verify files is to inspect Prusti for performance bottlenecks using a profiler, in order to identify and fix the most important issues.

3. **Allow users to choose whether to verify the current file or its enclosing module/crate**
   Currently, Prusti Assistant can only verify single files, which breaks down in most projects, seeing as they are usually split across multiple files. A workspace can have multiple modules & crates, so there should be an easy way to specify quickly exactly which module or crate you want to verify.

4. **Allow users to verify multiple crates with one command**
   Similarly to unit tests, it would give great peace of mind to run Prusti over a whole workspace and be sure that every file verifies. Enabling this with a single command and displaying the results appropriately will be helpful to users.

References

[1] V. Astrauskas, P. Müller, F. Poli, and A. J. Summers. Leveraging rust types for modular specification and verification. In Object-Oriented Programming Systems, Languages, and Applications (OOPSLA), volume 3, pages 147:1–147:30. ACM, 2019.

[2] Rust Programming Language. `https://www.rust-lang.org`. Accessed 2020-02-11.

[3] Peter Müller, Malte Schwerhoff, and Alexander J Summers. Viper: A verification infrastructure for permission-based reasoning. In International Conference on Verification, Model Checking, and Abstract Interpretation, pages 41–62. Springer, 2016.

[4] Visual Studio Code. `https://code.visualstudio.com`. Accessed 2020-02-11.

[5] Viper IDE. `https://bitbucket.org/viperproject/viper-ide`. Accessed 2020-02-11.

[6] Node.js. `https://nodejs.org`. Accessed 2020-02-11.