CSE 291K Research: Helping Humans Build Rusty Systems

Molly MacLaren

December 2024

Abstract This paper summarizes my research work for Fall 2024 in CSE 291K, focused on laying the groundwork for the development of pedagogical tools for programming in Rust. In particular, I develop research methods to assess whether functional-style Rust is truly "Rusty"—that is, if it is preferable and comprehendible by a diverse set of programmer demographics who use Rust. I detail the work leading up to this quarter, my current study design, and a side-quest where I re-framed a case study on Rust formal verification tools.

1 Introduction

In wake of catastrophes such as the global Crowdstrike outage last summer, software engineers working on critical systems need tools and methods to ensure that their programs will run safely. One such method is to use the Rust Programming language. Rust provides safety guarantees such as ownership rules for memory management, but these features are often difficult for programmers to adapt to [1].

My central research goal is to design tools and methods to make programming in Rust easier, without abstracting away the functionalities of the language that make Rust fast and secure. Through development of pedagocical learning tools, rather than short-term productivity enhancement tools, we can make the Rust programming language accessible to all types of programmers working on all types of software. In order to design tools that are helpful to a range of programming experiences, we must first understand common practices and mistakes among novice Rust programmers and programmers coming from other languages. I propose answering the question: "How does previous experience in Rust and other languages influence how much programmers use and comprehend functional-style Rust?"

This quarter, I worked on the development of my VS Code extension Situationally Adaptive Language Tutor (SALT). In previous work, I created this extension to host public-facing programmer studies, providing the user with a participation form and a demographics survey. In the extension, I designed a compiler error logging system that collects details such as error codes, affected lines, and hint messages without gathering user code, which may not be open source (Section 2). This quarter I began planning and implementing a study for analyzing functional versus imperative code usage in Rust alongside quiz and code review questions about functional and imperative design (Section 3). Additionally, I worked on revisions to a case study paper assessing the usability of Rust verification tools (Section 4).

2 My Previous Work

SALT or Situationally Adaptive Language Tutor is a VS Code extension I designed as a fork of REVIS [2], a visualization tool for Rust ownership and borrowing errors. In the evaluation of REVIS, I worked on recompiling states of student programs that were automatically committed to a repository on every save action. I identified areas for improvement in this study, including onboarding issues causing unrecorded data, lack of interaction tracking for interventions, and the limited scope of university participants. Thus, by integrating the study component *within* the VS Code extension, it is able collect the data and IDE inferences necessary for other types of analyses and expand to a larger participant-base of all Rust programmers and all Rust projects!

3 My Main Work

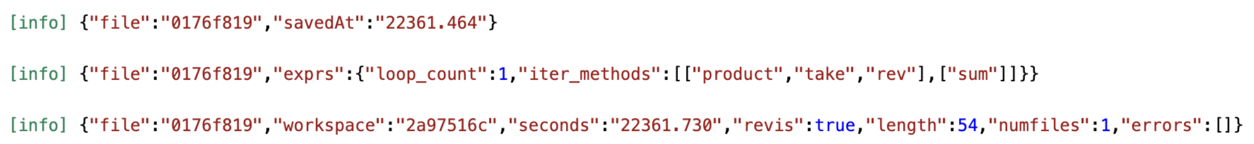
This quarter, I wanted to assess design patterns that are deemed *idomatic* in Rust. These idioms are stylistic guidelines to writing code that is "Rusty." Rust is a multi-paradigm language, but it is commonly claimed that programmers should use iterator functions whenever possible over imperative looping logic as they are more efficient [3]. Rust attracts systems programmers coming from memory-unsafe languages such as C and C++ which do not have these features. Additionally, some iterator functions such as `scan`, `fuse`, or `enumerate_back` are uncommon and their functionality may be obscure to programmers. As a result, these functional-style paradigms may be less readable to many Rust programmers, especially those who haven't programmed in a functional language such as Haskell before.

This quarter, I worked the infrastructure for designing a study to test this hypothesis. We designed two main methodologies:

1. Gather inferences from the HIR to determine how often programmers use loops and which types of iterator functions (Section 3.1).
2. Deploy quizzes and code review questions to programmers to test their knowledge and A/B test readability of imperative and functional Rust (Section 3.2).

3.1 Determining Usage

As part of the study onboarding, I included a simple demographics quiz which includes questions on how long the participant has been programming in Rust and other languages such as C, Javascript, and Haskell. To analyze how often different programmer demographics use functional versus imperative Rust styles, I used TypeScript with the VS Code API to call a custom Rust crate, `salt-ide`, to detect these expressions in the High-level Intermediate Representation (HIR). This crate emits JSON output which is read back in by the VS Code extension and logged. Flowistry is a similar tool for Rust which uses MIR data to display information flow visualizations [4]. Using Flowistry's methodology of calling Rust binaries from the extension side greatly simplified the FFI work.

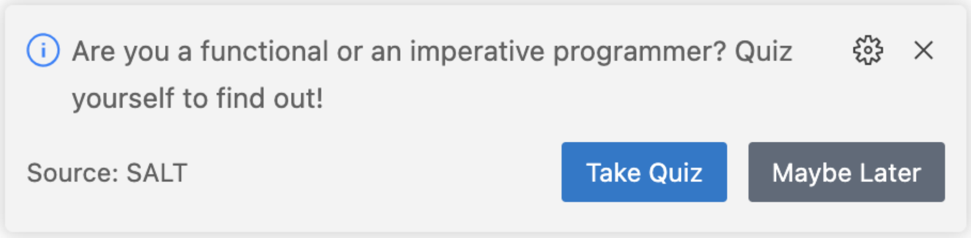


```
[info] {"file":"0176f819","savedAt":"22361.464"}

[info] {"file":"0176f819","exprs":{"loop_count":1,"iter_methods":[["product","take","rev"],["sum"]]}}

[info] {"file":"0176f819","workspace":"2a97516c","seconds":"22361.730","revis":true,"length":54,"numfiles":1,"errors":[]}
```

Figure 1: Output which is logged, given the programmer has one loop, a chain of iterator methods, and a single iterator method, viewable from the extension.



Are you a functional or an imperative programmer? Quiz yourself to find out!

Source: SALT

Figure 2: Sample notification within the extension.

With use of `rustc_plugin` and the nightly toolchain of Rust, working with the `rustc` API was possible without the need to fork `rustc` itself. The user does not need to be actively compiling with nightly either; only the extension must call `cargo salt-ide` with the toolchain. Thus, on every save action, the extension spawns a child process to detect whether `salt-ide` is installed with the correct nightly toolchain. If it is, then it makes a `cargo` call to the binary, and `cargo install salt-ide` if it is not.

On the Rust side, I used the HIR API to visit `ExprKind` enums in the current program. It collects a count of `Loop` expressions and chains of `MethodCall` expressions that are performed on types that implement the `iter` trait. This data is then formatted into JSON and printed back to the extension, where it is logged alongside save interval times and error diagnostics. An example of a log entry can be found in figure 1.

3.2 Quizzes and Tests

In the second part of this study, we plan to design short quizzes to test programmers' knowledge on iterator functions and imperative logic, and to assess the readability of these two paradigms. The quiz would be accessible after the participant has completed the study onboarding. The participant would be prompted with a notification which would link to a Qualtrics quiz, encoded with their participant ID (See figure 2).

First, the participant would be randomly given either an iterator function or loop implementation of a program and asked to determine the output. A simple example of several versions of the same program is in figure 3. Additionally, the participant would be asked one or more open-ended code review questions. These code review questions would involve code in either paradigm with intentional errors.

While my undergraduate research partner has taken most of the lead in this part, I plan to have a greater

```rust
fn main() {
  let numbers = vec![1, 2, 3];
  let mut result = 1;

  for num in &numbers {
    result *= num;
  }

  println!("{:?}", result);
}
```

```rust
fn main() {
  let numbers = vec![1, 2, 3];

  let result = numbers
    .iter()
    .fold(1, |acc,&num| acc*num);

  println!("{:?}", result);
}
```

```rust
fn main() {
  let numbers = vec![1, 2, 3];

  let result: i32 = numbers
    .iter()
    .product();

  println!("{:?}", result);
}
```

Figure 3: Imperative vs. Functional (fold) vs. Functional (product)

role in directing this process now that the quarter is over and I have more time to dedicate to research. Our question set is almost ready for pilot testing, so that we can assess their clarity and quality and to determine how many questions we should deploy to fit into a short and realistic time frame for participants.

I am planning to continue this work through this upcoming winter break and next quarter. We plan to deploy this study by the end of the year, collect data through the month of January, work on analysis during February, and submit a paper to the second round of OOPSLA 2025 in March!

4 My Related Work

Over the previous summer, I did research with the folks at Lawrence Livermore National Laboratory (LLNL) in the Center for Applied Scientific Computing. I worked on formally verifying Rust code, particularly the underlying union-find data structure and algorithm for *egg: e-graphs good* [5]. As my first experience with formal verification, and originally tasked to use the *Creusot* [6] verifier, I found that iterating between *Creusot* and *Prusti* [7] helped me develop a verification strategy. I also identified several areas where the Rust verifiers I used could improve with more IDE integration, documentation of proof debugging strategies, and clearer definitions for logical models. These enhancements could better align with Rust programmers' expectations in debugging and reasoning. Motivated by these reasons and my experience, I submitted a verification case study with a discussion on usability to VMCAI 2025.

Later this quarter, I received feedback from the reviewers—and while the result was not what I had hoped—I earned some very valuable information about how this work should be presented. Rather than focus on explaining the results of the verification, I needed to provide more detail about the process; what were the most difficult aspects, and what did the development timeline look like?

I had just three weeks to re-frame this paper for another conference which I felt would be a better fit. FormaliSE focuses on the intersection of software engineering and formal methods. I ultimately re-wrote most of the paper as the page count was halved, and I wanted to add more sections. I gave better details about the functions which take a mutable reference to the union-find itself, as opposed to non-mutating functions which were fairly trivial to verify their execution. I also produced a timeline and highlighted the points at which I reached out to the tool developers, which was the greatest resource I had towards getting started and overcoming roadblocks.

However, asking the developers for help is not something that scales well! In my paper, I highlighted the need for resources on debugging strategies and writing supporting lemmas for unprovable SMT solver statements [8]. User guides should be more like learning resources or textbooks, rather than quick references, to make tools accessible to a broader audience beyond formal methods experts. Just as verification frameworks abstract away their inner workings, completed proofs hide the process of writing and debugging—something better demonstrated through tutorials or even videos!

References

[1] K. R. Fulton, A. Chan, D. Votipka, M. Hicks, and M. L. Mazurek, "Benefits and drawbacks of adopting a secure programming language: Rust as a case study," in *Seventeenth Symposium on Usable Privacy and Security (SOUPS 2021)*, pp. 597–616, USENIX Association, Aug. 2021.

[2] R. Wang, M. MacLaren, and M. Coblenz, "Revis: An error visualization tool for rust," 2023.

[3] S. Klabnik and C. Nichols, *The Rust Programming Language.* No Starch Press, first ed., 2016.

[4] W. Crichton, M. Patrignani, M. Agrawala, and P. Hanrahan, "Modular information flow through ownership," in *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI '22, p. 1–14, ACM, June 2022.

[5] M. Willsey, C. Nandi, Y. R. Wang, O. Flatt, Z. Tatlock, and P. Panchekha, "egg: Fast and extensible equality saturation," *Proc. ACM Program. Lang.*, vol. 5, Jan. 2021.

[6] X. Denis, J.-H. Jourdan, and C. Marché, "Creusot: a foundry for the deductive verification of rust programs," in *ICFEM 2022 - 23th International Conference on Formal Engineering Methods*, Lecture Notes in Computer Science, (Madrid, Spain), Springer Verlag, Oct. 2022.

[7] V. Astrauskas, A. Bílý, J. Fiala, Z. Grannan, C. Matheja, P. Müller, F. Poli, and A. J. Summers, "The prusti project: Formal verification for rust," in *NASA Formal Methods (14th International Symposium)*, pp. 88–108, Springer, 2022.

[8] M. MacLaren, E. Westbrook, J. Sarracino, and M. Sottile, "Usability lessons towards adopting deductive verification in mainstream rust development," in *Preprint*, 2025.