A Multimodal Study of Challenges Using Rust

Michael Coblenz [1], **April Porter** [2], **Varun Das** [2], **Teja Nallagorla** [2] **and Michael Hicks** [3]

[1] *University of California, San Diego, California* [a]
[2] *University of Maryland, College Park, Maryland*
[3] *Amazon and University of Maryland, College Park, Maryland* [b]

[a] Work performed at the University of Maryland
[b] Work performed prior to starting at Amazon

Abstract

Rust is a programming language that provides strong safety properties, but does so at a usability cost. We conducted an observational study of Rust learners and a thematic analysis of StackOverflow posts about Rust to identify opportunities for improvement in Rust's usability. Key challenges that we observed included syntactic challenges posed by the `?` operator, block-terminal statements, and automatic dereferencing; late delivery of error messages; and the opacity of Rust errors resulting in programmers being unable to identify whether their partial fixes represented progress. We describe a collection of opportunities for improvement that leverage the compiler and the IDE.

Keywords: Rust. Usability of programming languages. Empirical studies of programmers.

1 Introduction

The Rust programming language is gaining popularity among many different kinds of programmers, having been ranked "most loved" for seven years [1]. However, Fulton et al. [2] found that 59% of survey respondents felt Rust was *harder to learn* than other programming languages. Although Rust provides strong safety properties, such as memory safety, programmers (and users of their programs) will only reap the benefits if programmers learn the language successfully. If we could identify ways of making Rust easier to learn and use, perhaps more software teams will adopt Rust, making software safer for users.

In this work, we took two different approaches to understand which aspects of Rust are most challenging for programmers. To understand challenges *learners* face, we conducted a think-aloud study of students who were taking a Rust programming class in which we observed students working on their programming assignments. We found that beginners struggled with the implications of automatic dereferencing; with the semantics of the `?` operator; with error messages that were delivered after they were needed or which were too hard to read and understand. Also, as programmers tried to fix errors, they sometimes could not tell whether their changes represented progress. This challenge may have been exacerbated by the complexity of the type system and the borrow checker, which analyzes safety of references. These observations also provide insight into students' programming processes and students' reliance on tools like StackOverflow, YouTube, and online documentation in addition to course materials such as lecture notes. We also observed the debugging and problem-solving practices the students used, such as desugaring syntactic sugar.

To see how our results might generalize to the broader Rust community, we also conducted a thematic analysis of the 646 most frequently viewed questions in the Rust category on StackOverflow [3]. We had expected that challenges with ownership and borrowing would dominate the discussion, since these features are unique to Rust and have been reported to present a significant learning curve for programmers. Although we found that difficulties with ownership and borrowing were well-represented, we also observed a broader set of challenges with the Rust standard library and data structures as well as with Rust's abstraction mechanisms. These challenges are likely more representative of the issues faced by more-expert Rust programmers who work on larger projects.

2 Observational Study of Rust Learners

We were interested in understanding what obstacles new Rust programmers face. In order to observe learners, we collaborated with two graduate students, who taught a course on Rust. The course

PLATEAU
13th Annual Workshop at the Intersection of PL and HCI

Organizers:
Sarah Chasins, Elena Glassman, and Joshua Sunshine

had sophomore-level discrete math and computer systems courses as prerequisites, so the students had already learned C in the context of systems programming. Our research approach was to recruit students for think-aloud observation while they worked on their assignments. We offered 2% course extra credit for each one-hour observation period up to three observations per student. After the observations were complete, the experimenter provided help with Rust issues that the students had encountered. This approach provided educational value to the students and offered an opportunity for the experimenter to obtain additional insight into the challenges. We recorded videos of the screens and audio of the discussion and we took notes regarding challenges that the students encountered. We view each challenge as a possible opportunity for improvement in the language, IDE, or educational setting. Therefore, rather than conducting a structured analysis of the transcripts, we report in this paper on the challenges we observed that could have implications on language design, tool design, or pedagogy.

During the sessions, we provided help with building and running the project as necessary. We also probed to understand the participants' work strategies more deeply, asking questions such as "Why did you X?" (where "X" is something the participant did), or "What's the question you're thinking about?" when the participant appeared to be confused. We also found "If you were to ask a question, what would it be?" helpful in elucidating which points participants found confusing.

We conducted sixteen observations during the semester. Because the observations spanned five different homework assignments as well as an independent project, the observations offered the opportunity to see challenges that students faced in a variety of different contexts. It also enabled us to observe challenges at different levels of experience as the course progressed. Although this approach did not enable us to collect statistics regarding how often students faced each challenge, it provided a broad perspective regarding opportunities for improvement in the language, tools, and pedagogy. Some of the challenges may be common challenges faced by earlier-stage programmers regardless of language. Nonetheless, they are worth discussing because the challenging setting of Rust may have exacerbated some of these problems. To preserve our participants' anonymity, we identify participants in this paper with identifiers such as *P1*.

The assignments we observed covered a variety of topics, including ownership, structs, pattern matching, error handling, traits, smart pointers, interior mutability, closures, and concurrency. Details of the schedule are available online[1].

2.1 Matters of Rust language design and errors

The `?` operator In Rust, the `?` operator can be used on an expression of type `Result` to cause the containing function to return an error if the expression itself represents an error. Of course, this only works on functions that themselves have return type `Result`. P3 used the `?` operator in a function that did not return `Result` and saw the error message `Cannot use the ? operator in a function that returns '()'`. Unfortunately, P3 did not realize that the error was about the context of `?` rather than the expression to which it was applied, and spent about a minute and a half manually desugaring the use of `?`.

Rust's compiler has a reputation for having particularly good error messages [2]. However, even correct, clearly written error messages can include extraneous information, which readers can find confusing. P4 observed the error message: `` the '?' operator can only be used on `Option`s, not `Result`s, in a method that Returns `Option`; use `.ok()?` if you want to discard the `Result<Infallible, std::id::Error>` error information ``. P4 followed the suggestion, appending `.ok()?`, but commented "I have no idea what this is", highlighting the `Infallible` type. The `Infallible` type, with which the participant was unfamiliar, was not mentioned in the code, so P4 was surprised to find it. P13 encountered a similar problem: an error message referred to traits, with which P13 was unfamiliar, leading to spending several minutes looking for information that was not relevant to the actual problem.

1 https://www.cs.umd.edu/class/fall2021/cmsc388Z/

Block-terminal statements and expressions In Rust, the value of a block is the value of the block's last expression. When the block ends in a statement, the statement is considered to have type `()`. Also, Rust has `if` *expressions* rather than `if` *statements*. Some participants found this confusing. Take as an example:

```
1  fn test() -> i32 {
2      let x = if (...) {
3          42; // programmer intended the 'then' branch to evaluate to 42
4      }; // else branch omitted for brevity
5      x
6  }
```

The above function, which has a superfluous `;` after `42` on line 3, gives this error message:

```
4 | fn test() -> i32 {
  |              --- expected `i32` because of return type
...
8 |     x
  |     ^ expected `i32`, found `()`
```

P5 found this behavior confusing — what is `()` and how does the code produce it? Statements, which end in semicolons, have type `()`; the programmer needed to remove the `;` after `42` to make it an expression rather than a statement. Although support for `if` *expressions* rather than if *statements* can be convenient at times, in this case it resulted in a surprising error message.

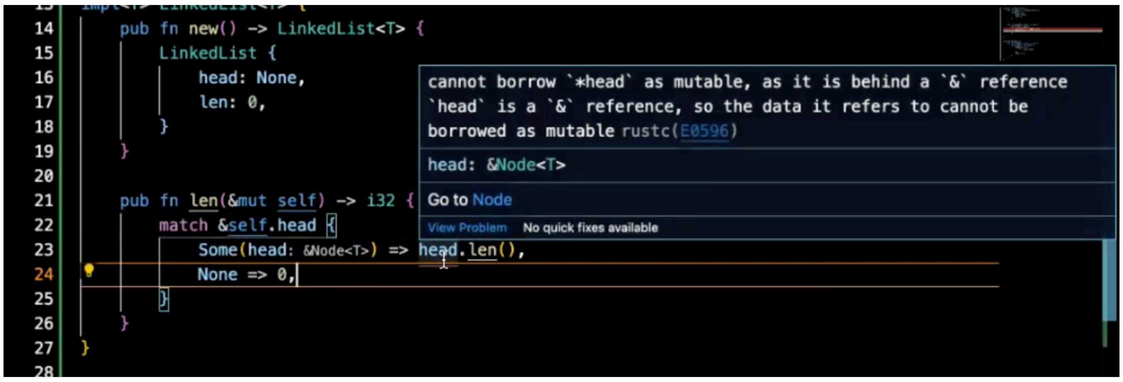
Automatic dereferencing, deref coercion, and mutable references In Rust, the `*` operator dereferences references. However, in certain situations, the `*` operator is implicitly inserted by the compiler — for example, in `match` expressions[2] and when the `.` operator is used (e.g. one can write `x.foo()` rather than `(*x).foo()`)[3]. In addition, when a value implements the `Deref` trait, which indicates that the `*` operator can be used to dereference values of a type, the *deref coercion* feature[4] means that the compiler automatically inserts the `*` operator as many times as is necessary to convert references between types — but only for arguments to function and method calls.

The rules are complicated, and some participants were confused about when to use `*` and `&`. P7 said: "I'm pretty bad with understanding when I need to borrow with `&` and when I don't." Figure 1 shows one example from P7's work in which P7 needed to add `mut` in a `match` expression. The fact that `&` is used both as a unary operator to get a reference to a value *and* as a type operator to indicate that a type is borrowed, may complicate understanding. Although this design is in common with C++, based on the curriculum, our participants likely had no C++ experience. P14 thought unary `&` would cause a move, when in fact it would cause a borrow. P16 briefly tried to use `&` to get the value *out* of a `Box`, as if it were a dereference operation (`*`).

Participants also relied on the compiler to tell them when values needed to be mutable. After seeing an error, `cannot borrow *head as mutable, as it is behind a `&` reference`, P7 remarked, "I really rely on these messages it gives me. They're extremely helpful." Nine minutes later, the same problem came up again: he was calling a method that took a `&mut` receiver and was confused as to why the argument of a `match`, on which he was invoking the method, also needed to be mutable. "I'm very confused about these...". The situation highlights the fact that language design decisions that optimize for concision can cause a steep cost in usability.

Interior Mutability Interior mutability is Rust's mechanism for moving memory safety checks from compile time to run time when necessary to relax aliasing constraints. Prior work [4] focused on using a garbage collector to mitigate learning challenges posed by interior mutability. In this assignment, participants used traditional Rust, without a garbage collector. One challenge that P11 faced was related to the fact that using interior mutability to obtain multiple mutable references requires understanding *two* different structures: `Rc`, which implements a reference-counted smart pointer, and

2 https://github.com/rust-lang/rfcs/blob/master/text/2005-match-ergonomics.md
3 https://stackoverflow.com/questions/28519997/what-are-rusts-exact-auto-dereferencing-rules
4 https://doc.rust-lang.org/book/ch15-02-deref.html



```
14      pub fn new() -> LinkedList<T> {
15          LinkedList {
16              head: None,
17              len: 0,
18          }
19      }
20
21      pub fn len(&mut self) -> i32 {
22          match &self.head {
23              Some(head: &Node<T>) => head.len(),
24              None => 0,
25          }
26      }
27  }
28
```

```
cannot borrow `*head` as mutable, as it is behind a `&` reference
`head` is a `&` reference, so the data it refers to cannot be
borrowed as mutable rustc(E0596)
head: &Node<T>
```

Figure 1. P7 was unsure when values needed to be mutable and needed to add `mut` on line 22 after `&`.

`RefCell`, which moves the safety check from compile time to run time. P11 read documentation and concluded, "`borrow_mut` allows us to access what's inside `Rc`," but `borrow_mut` is a method on `RefCell`, not `Rc`. The confusion may have been driven by Rust's automatic dereferencing feature, which enables accessing methods on objects referenced by smart pointers (such as `Rc`) without the `*` operator.

Debugging implicitly-invoked methods P11 wrote a plausible-looking linked list implementation, but the test cases failed to terminate. P11 tried to debug the situation for 15 minutes by guessing possible causes, none of which was right. At the end of the session, the experimenter provided debugging assistance, which revealed that the problem was a faulty `drop` implementation. Because the Rust compiler automatically inserts calls to the `drop` method, debugging the situation was challenging, since the infinite loop appeared to happen *after* the last line of code of a function.

2.2 Strategic and Workflow Challenges in Rust

Ignoring error message content Some participants, including P5 and P11, noticed red underlines in their IDE (Visual Studio Code), indicating that the compiler had emitted errors for that code, but did not bother to read the accompanying error messages. Instead, they tried to guess what the problem was on the basis of the portion of code that was underlined. Of course, we cannot infer whether this strategy was beneficial overall relative to taking time to read the error message. However, this finding contrasts with that of Barik et al. [5], who found that, in aggregate, developers generally *do* read error messages. Although it is possible that this behavior is not specific to Rust, the wide range of possible problems in Rust (and beginner's difficulty in guessing them) may mean that ignoring error message content is particularly problematic in Rust.

2.3 The Process of Learning Rust

Success despite partial understanding P8, who was implementing `push` on a linked list, wrote code to create a new node and update the head structure to point there. He was confused about whether the new node was on the stack or the heap. He concluded that the node must be on the heap, since otherwise the result would be unsafe, even though the code looked as though it was putting the data on the stack. In fact, because he was assigning to a field of `self`, the location depended on the location of `self`, which was not apparent locally. Interestingly, although this is a fundamental question about Rust memory management, P8 managed to write appropriate code without understanding the semantics.

2.4 Opportunities for Many Languages

Errors delayed are errors denied Nielsen's heuristics [6] include *visibility of system status*; an implication of this is that IDEs that show error messages should display the error messages right away when the code becomes erroneous. This corresponds with results from live programming [7], which tries to always show the results of programs[5]. Unfortunately, some participants experienced a delay between edits and display of errors (as is common, since compilation is not always instantaneous, especially on older hardware). Some participants wrote erroneous code, did not receive an error, concluded their code was valid, and scrolled to a different place in the file to continue their work. As a result of the error (perhaps a syntax error, which precludes further error detection), the participants continued writing code, believing their new code was valid when in fact it had serious problems. P5, who encountered this problem, then had to context-switch back to the previous task in order to correct an error that had been inserted much earlier.

At 49:21, P8 inserted a superfluous parenthesis, causing a syntax error. The IDE displayed a red underline under the parenthesis, but P8 apparently did not notice the underline. At 51:40, P8 expressed surprise that the IDE did not report any errors in the new code. The error persisted until 54:03, when P8 found the extra parenthesis. For nearly five minutes, then, the participant did not receive live feedback on the new code. Fixing the error resulted in five new errors that needed to be fixed.

One implication of Rust's type system design is that the borrow checker cannot be run until after type errors have been corrected. Unfortunately, this behavior annoyed P10. P10 corrected a consistent mistake that she'd made in several methods, and then obtained new errors in three places pertaining to borrowing. While working, P10 remarked, in an increasingly frustrated tone: "How is what I'm doing in *this* function affecting what the compiler's saying about *this* one? That's some, like, you know…I'm just going to comment this one out…I don't understand …why does it not give me the errors…that's really obnoxious." Generally, programmers do not need to understand implementation details of their compilers; expecting that programmers know under what conditions the borrow checker runs was detrimental for some participants.

Small mistakes have high costs While writing a function, P6 believed he needed to convert a `Vec` of `Result`s to a `Result` of `Vec`s. He searched online, which took him to a StackOverflow page, which advised to invoke `into_iter().collect()`. Unfortunately, this resulted in a lengthy error message:

```
a value of type 'std::result::Result<std::vec::Vec<regex::Regex>, &str>' cannot be
    built from an iterator over elements of type 'regex::Regex'.
70 |    let ans: Result<Vec<Regex>, &'static str> = x.into_iter().collect();
                                                                  ^^^^^^^ value of type
    `std::result::Result<std::vec::Vec<regex::Regex>, &str>` cannot be built from
    `std::Iter::Iterator<Item-regex::Regex>`.
 - help: the trait `std:iter::FromIterator<regex::Regex>` is not implemented for `std::
     result::Result<std::vec::Vec<regex::REgex>, &str>`
```

P6 first (incorrectly) thought the problem was that the vector could not be iterated. "Maybe I can do `x.collect()`…" Removing the `into_iter()` call resulted in an even more confusing error about unsatisfied trait bounds. P6 guessed that the problem was that the variable in question was not mutable, but fixing that resulted in the same error. Then he believed that the problem was that a vector is not an iterator, which was true, and he reverted to the previous code. Then he started trying to understand the original StackOverflow post by reading documentation for `collect()`. Then he thought perhaps the problem was that he'd used `iter()` originally rather than `into_iter()` even though in fact he *had* used `into_iter()` initially. Then he tried adding a type annotation when constructing the vector, but type annotations go on declarations, not constructor invocations, so he got an `expected expression` error. "It's set up like the [example] on the page, but it doesn't act like it." Then he studied documentation for the `Result` type. Finally, he focused on trying to understand the type of the result of calling `collect`, and 53 minutes after doing the initial search for how to convert Vector of Results to a Result, he discovered that he *already* had a vector of Regexes and merely needed to wrap it in a `Result`.

Typically, we think of debugging in terms of understanding run time behavior of programs. In this case, the process of understanding the compiler's reasoning was akin to debugging: P6 generated

5 Will Crichton points out that it can be unclear when to display errors, since it can be annoying to show errors on partially complete code that the author already knows is incorrect

hypotheses for the problem and repeatedly ruled them out, necessitating generating *new* hypotheses. When a programmer writes code that embodies incorrect assumptions, it can require substantial skill to use the compiler errors to identify the incorrect assumptions.

Error-dense program spaces P13 had trouble telling whether his attempts at fixing compiler errors were making progress, since each of several changes in sequence led to a new compiler error. Both *progress* and *no progress* can result in different errors. Eventually, P13 started forgetting which approaches he had tried, needing to re-try approaches he'd already tried.

Videos instead of documentation P8 was working on a linked list implementation in an assignment that focused on heap allocation using `Box`. He was confused about why the wrapper, `LinkedList`, used an `Option<Node<T>>` while the `Node` struct has an `Option<Box<Node>>`. He located a YouTube channel, *Let's Get Rusty,* which he'd previously found, and found a video on smart pointers. After spending five minutes watching the video, P8 looked online for information about cloning `Box` objects, and concluded that `Box` was like `malloc`, which he already understood from C. Compilers give error messages in textual form. If some users find that videos are preferable for explaining important concepts, compilers could provide videos that explain error messages or that teach related concepts, and IDEs could facilitate viewing them without a separate search.

3 Analysis of StackOverflow Questions

In the previous section, we described our observations of students doing programming tasks in a Rust class. To gather data about a broader population of Rust programmers, we obtained IRB approval and conducted a thematic analysis [8] of questions about Rust on the StackOverflow question-and-answer web site [3]. Rust is popular among StackOverflow users, who ranked Rust as their most loved language for seven years [1]. We were particularly interested in learning what challenges pertained to the type system, since that aspect of the language could be applicable to other language designs as well. We were also interested in StackOverflow because it offered a broader sample of programmers than our student population: 80% StackOverflow users have 5 or more years of development experience. We downloaded the most frequently *viewed* questions as of as of March 7, 2021. Then, we iteratively coded batches of 50 questions per coder in order of decreasing view count until we found that new codes in batches were infrequent, coding 646 questions total. The 646 questions had been viewed a total of 11.4M times. The top 2000 most-viewed questions had accrued a total of 14.7M times, so our analysis covered 78% of the views among the top 2000 questions.

Three student researchers coded the questions individually, raising unclear StackOverflow questions for group discussion and resolution with a faculty co-author. The wide variety of questions resulted in 82 different codes. To evaluate consistency, all three coders coded the same 50 questions at the end of the process. Krippendorff's alpha was 0.90, indicating strong agreement. Finally, a faculty co-author spot-checked codes to improve quality.

Table 1 shows the eight themes we identified that cover the most commonly-used codes. These themes cover 7.7M of the 11.4M views among the questions we coded. The diversity of topics compared to the comments in the experiment survey likely represents the much broader and deeper use cases of the general population of Rust programmers, who need to use complex data structures and leverage many different components of the standard library.

The large number of views of basic questions about ownership, references, and lifetimes provides evidence that some of the issues that we observed in the study pertain not only to university students but also to programmers in general. Ownership, references, and lifetimes feature prominently among the most-viewed questions (948K views). Question #34 (46K views), for example, was "What do I have to do to solve a 'use of moved value' error"? Question #89 (31K views) was "Why can't I store a value and a reference to that value in the same struct?" Most ownership questions were from Rust beginners who were trying to express familiar concepts in existing languages. For example, one user (#258, 14K views) was trying to manipulate a linked list (a notoriously tricky problem in Rust [9]) and was unsure how to fix ownership errors. Another (#268, 13K views) was trying to relate move semantics as in C++ to Rust ownership. A third (#172, 19K views) was writing accessors but was confused as to how to do it without destroying the owned fields that were being accessed.

4.4M views pertained to usage of specific structures or libraries. This is not particularly surprising, since the standard library is large and the collection of applications diverse.

4 Discussion and Future Work

4.1 Comparing StackOverflow results to course-based observations

The StackOverflow analysis shows that many of the challenges that Rust programmers face pertain to the details of data structures and the standard library in general. This is not surprising, considering the large API surface involved and the many design questions involved in their creation. However, consistent with the observations we conducted, ownership and lifetimes also pose significant challenges.

Since the participants in our observations were Rust novices, the StackOverflow posts may represent challenges faced by more-experienced programmers. However, we were surprised by the large number of questions (860K) pertaining to structs and traits, which are fundamental features in Rust. We were also surprised by the large number of questions about type conversion and numeric operations, since those are relatively low-level issues. One possibility is that the complexity of Rust's type system results in more situations in which equivalent data have different types and thus must be converted.

One aspect of the StackOverflow results that was unclear is which of the StackOverflow questions are frequently viewed because it can be more convenient to browse StackOverflow than to read the manual, and which questions are sources of significant challenges that the manual cannot or does not address adequately[6].

4.2 Opportunities for tools

Concision vs. Explicitness When language design choices involve trading off concision and explicitness, concision can result in confusing syntax [10]. The Rust operator `?` provides a concise (but not very explicit) way of handling values of type `Result`. The `?` operator, however, can only be used in functions that have return type `Result`, with the choice to make `return` unnecessary at block-terminal positions, and potentially with the *deref coercion* feature. It remains to be seen how to add these features that improve concision without compromising clarity, particularly for language novices. One possible approach might involve IDE features that optionally (and temporarily) expand syntactic sugar so that programmers could understand the semantics in terms of more common or simpler operators. Another approach is that of Tutorons [11], which provides explanations when the user clicks code for which Tutorons supports explanations.

The challenges P3 and P4 faced with the `?` operator likely reflects problems both with the design

6 Thanks to Will Crichton for pointing out this distinction.

Table 1. Themes identified in a thematic analysis of most-viewed StackOverflow questions about Rust. The second column shows the number of views of questions with codes related to each theme.

Theme	Views	Description
Data structures	3.1M	Slices, tuples, arrays, array operations, strings, hash maps, vector operations
Libraries	1.4M	Standard and third-party libraries, files, time, printing/formatting
Abstraction	860K	Structs and traits
Type Conversion	790K	The `as` cast operator, vector to array conversion, enum to integer conversion
Ownership	678K	Ownership and its semantics, referencing, `Box` (`Box` questions frequently related to ownership)
Iteration	519K	Creating and using iterators
Lifetimes	270K	Lifetimes and lifetime parameters
Numbers	190K	Numeric operations

of the `?` operator and with the error message. The `?` operator is certainly inconsistent with other operators in that it has non-local effects. The confusion that resulted from the error message is interesting because the error message was correct, but the participant interpreted it incorrectly.

Currently, tools expect programmers to choose of various possible ways of expressing a program, and sometimes offer refactoring features that can transform programs in some cases. If there were a body of evidence regarding the usability attributes of different ways of expressing a program, and transformation operators that could transform among them, then an IDE could help make code more usable by recommending and executing appropriate transformations. The first step is to leverage and strengthen evidence for particular languages (Rust, in this case) that would enable building such a tool.

Surprising invocations Language runtimes, as well as frameworks, are often designed to invoke functions at times that the authors do not expect or predict. In this study, P11 found it difficult to figure out that their `drop()` implementation was responsible for non-terminating behavior because the call to `drop()` was inserted by the compiler. What if IDEs understood the semantics more deeply, so that programmers could see the automatically inserted code as well? Likewise, IDEs usually are not sufficiently integrated with frameworks to be able to show in what contexts a function will be called. This can make it hard to understand the implications of a given change.

Block-terminal statements vs. expressions The confusion that P5 experienced when an expression had an extra semicolon was related to the fact that statements have type `()`. Perhaps a better error message would have helped[7]; after all, the particular expression that was used as a statement could have been shown to have no side effects and therefore the semicolon was surely a mistake. Or perhaps it would help if the IDE emphasized that a statement had been provided where a expression was expected.

Emphasizing off-screen error messages IDEs show errors when the errors become available. Due to compilation time, there is often a delay between writing erroneous code and detecting the error. Unfortunately, the user's attention may have focused on a different part of the program. What if IDEs visually emphasized off-screen errors whose presence had implications on the present code? Then users could be more confident that any analysis results that were being displayed were accurate — and make appropriate fixes if not.

References and borrowing: the hard parts of ownership? We did not observe students struggling with the *concept* of ownership. Instead, what they struggled with was the semantics of references and borrowing. This difficulty may have been exacerbated by Rust's implicit `*` and the fact that `&` is overloaded as an operator on both expressions and on types, with slightly different meanings. Crichton [12] argued that understanding the borrow checker and its limitations are significant challenges for Rust learners. The problems we observed were, in some sense, earlier stage: first one should understand what the operators do before trying to write code and understand how the compiler arrived at its conclusions.

Making errors easier to understand Adding a visual representation of error messages might make some errors easier to understand. For example, *use after move* errors might be clarified with diagrams provided by tools such as RustViz [13]. If the errors were easier to understand, users might be more likely to read them (rather than guessing the problem based on the presence of the red underline) and faster to fix the underlying problems.

Tracking debugging attempts Some participants tried a long sequence of techniques to try to fix compiler errors, eventually forgetting which techniques they had already tried. Perhaps IDEs could help programmers keep track of their problem-solving attempts so far, rather than assuming

7 We filed a bug: https://github.com/rust-lang/rust/issues/105431

programmers will do so on their own. This might be helpful both for traditional debugging and for resolving complex compilation problems in Rust.

A pedagogical IDE Typically, IDEs report exactly the errors given by compilers, which always give the same error messages for a given erroneous situation. However, suppose an IDE kept track of the programmer's expertise by observing their programming process and their history of errors, akin to a *cognitive tutor* [14]. Could an IDE provided more instructive error messages when recognizing conceptual gaps? Error messages could encourage the programmer to read conceptual documentation or watch video explanations if understanding the error message required knowledge of concepts the programmer had not yet mastered. If a programmer receives the same error message on the same line of code several times in a row, despite making attempts to fix the error, the IDE could provide metacognitive assistance by recommending alternative strategies for resolving the problem.

5 Related Work

Prior work [4] described a randomized controlled trial of Bronze, which is a prototype garbage collector for Rust. We found that a garbage collector can reduce times by novice Rust programmers by a factor of three for tasks that require using interior mutability in a way that impacts the architecture. For other tasks, there was no significant difference in task completion times. Notably, although participants who used the garbage collector did not like Rust more than other participants, they believed the garbage collector was helpful. Clearly, the architectural choices in Rust projects (which are in some cases driven by type system constraints) have implications on the ability of programmers to be successful.

Fulton et al.[2] described an interview and survey study of developers who had considered using Rust. They found that many participants had experienced a steep learning curve when first using Rust. As a result, the participants were concerned about using Rust in real software projects: Rust is a new enough language that new team members would need to learn Rust, but learning Rust is expensive, so the costs of getting people ready could be very large. This forms a barrier to Rust adoption in practical settings. However, if we could make Rust easier to learn and use, the adoption barrier could be much lower. Our work focuses on clarifying why Rust might be hard to learn and use and on identifying opportunities for improvement.

Crichton [12] argued that a significant cause of usability challenges in Rust relates to the borrow checker. The borrow checker's analysis must be sound, so it is necessarily incomplete, and users may struggle to compile programs that are sound but for which the borrow checker cannot prove safety. Thus, in order to program in Rust, programmers must understand the ownership rules, how the borrow checker works, and how to work around the limitations of the borrow checker. In this paper, we focus on challenges to Rust usability in addition to those posed by the borrow checker. Both we and Crichton propose work on visualizations and on better error messages.

Zhang et al. [15] conducted a corpus study of StackOverflow, finding that 31% of the suggestions included potential API usage violations. Many participants in our study used code from StackOverflow and other sources without attempting to understand it in depth. Future work should evaluate whether languages with more complex, hard-to-understand compiler analyses may tend to result in more incorrect usage than would occur in other languages.

Zhu et al. [16] also studied StackOverflow posts about Rust, analyzing 15,509 posts using an automated LDA model; then, they randomly sampled 100 posts and analyzed those manually. Our approach involved manually analyzing the 646 most-viewed questions; we selected this approach because we felt that we needed to capture a large portion of the views to understand the landscape thoroughly. However, their findings were generally consistent with ours; they observed questions about library functions, Rust's type system, type conversions, and traits. They validated their findings with a survey in which they asked participants to identify errors in Rust programs. Participants had difficulty doing these tasks correctly, confirming that understanding and applying the rules of the Rust type system is challenging. They also identified several opportunites for improvement in specific error messages.

Zeng and Crichton [17] analyzed Reddit and Hacker News posts and comments regarding Rust and identified three possible impediments to Rust adoption: lack of clear messaging regarding the benefits of Rust tooling to users; challenges with discoverability of patterns for writing unsafe code when necessary to write complex library code; and the failure of Rust users to attempt incremental migration of their codebases to Rust. These broader software engineering challenges with using Rust were not likely to arise in the context of the tasks in our study, so this insight complements our results.

RustViz [13] provides visualizations for ownership and borrowing in Rust code. It requires that an author annotate code with comments, which the RustViz system leverages to draw diagrams. Nearly all the students reported that the visualizations were helpful. In the future, RustViz or similar tools could be used to address some of the challenges reading and using error messages that we identified in this study.

6 Conclusion

Leveraging two different data sources to identify usability challenges in Rust has helped us identify a collection of opportunities for improvement both in tools for Rust and in future language designs. IDE techniques could blunt the impact of language choices that favor concision over explicitness; help draw attention to off-screen error messages that are relevant but not delivered in a timely fashion; and provide diagrammatic explanations of error messages to make some errors easier to read.

Our analysis of StackOverflow questions shows that challenges with ownership and borrowing persist among a broader population of Rust users. In addition, these users face challenges using the standard library, despite the extensive accompanying documentation, as well as with using basic abstraction features such as structs and traits. Taken together, these data suggest that there are significant opportunities for improvement for both beginners and for a broader population of Rust programmers. By making Rust easier to learn and use, we may enable more software projects to obtain Rust's strong safety properties.

7 Acknowledgments

We thank Dongze He and Chase Kanipe for teaching CMSC388Z[8], Programming in Rust, at UMD. We also thank Will Crichton, our mentor at the PLATEAU workshop, for insightful suggestions on this work. This material is based upon work supported by the National Science Foundation under Grant No. 1801545. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

References

[1] Stack Exchange, Inc. "Developer survey results." (2022), [Online]. Available: https://survey.stackoverflow.co/2022/#section-most-loved-dreaded-and-wanted-programming-scripting-and-markup-languages.

[2] 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)*, USENIX Association, Aug. 2021. [Online]. Available: https://www.usenix.org/conference/soups2021/presentation/fulton.

[3] Stack Exchange, Inc. "Stackoverflow." (2021), [Online]. Available: https://stackoverflow.com.

[4] M. Coblenz, M. L. Mazurek, and M. Hicks, "Garbage collection makes rust easier to use: A randomized controlled trial of the bronze garbage collector," in *Proceedings of the 44th International Conference on Software Engineering*, ser. ICSE '22, Pittsburgh, Pennsylvania: Association for Computing Machinery, 2022, pp. 1021–1032, ISBN: 9781450392211. DOI: 10.1145/3510003.3510107. [Online]. Available: https://doi.org/10.1145/3510003.3510107.

[5] T. Barik, J. Smith, K. Lubick, *et al.*, "Do developers read compiler error messages?" In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, IEEE, 2017, pp. 575–585.

[6] J. Nielsen and R. Molich, "Heuristic evaluation of user interfaces," in *SIGCHI conference on Human Factors in Computing Systems*, ser. CHI 1990, 1990.

8 https://www.cs.umd.edu/class/fall2021/cmsc388Z/

[7] T. Lieber, J. R. Brandt, and R. C. Miller, "Addressing misconceptions about code with always-on programming visualizations," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ser. CHI '14, Toronto, Ontario, Canada: Association for Computing Machinery, 2014, pp. 2481–2490, ISBN: 9781450324731. DOI: 10.1145/2556288.2557409. [Online]. Available: https://doi.org/10.1145/2556288.2557409.

[8] V. Braun and V. Clarke, "Using thematic analysis in psychology," *Qualitative research in psychology*, vol. 3, no. 2, pp. 77–101, 2006.

[9] A. Beingessner. "Learn rust with entirely too many linked lists." (2021), [Online]. Available: https://rust-unofficial.github.io/too-many-lists/.

[10] M. Coblenz, G. Kambhatla, P. Koronkevich, *et al.*, "Pliers: A process that integrates user-centered methods into programming language design," *TOCHI*, vol. 28, no. 4, Jul. 2021, ISSN: 1073-0516. DOI: 10.1145/3452379. [Online]. Available: https://doi.org/10.1145/3452379.

[11] A. Head, C. Appachu, M. A. Hearst, and B. Hartmann, "Tutorons: Generating context-relevant, on-demand explanations and demonstrations of online code," in *2015 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, 2015, pp. 3–12. DOI: 10.1109/VLHCC.2015.7356972.

[12] W. Crichton, *The usability of ownership*, 2020. DOI: 10.48550/ARXIV.2011.06171. [Online]. Available: https://arxiv.org/abs/2011.06171.

[13] M. Almeida, G. Cole, K. Du, *et al.*, "Rustviz: Interactively visualizing ownership and borrowing," in *2022 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, 2022, pp. 1–10. DOI: 10.1109/VL/HCC53370.2022.9833121.

[14] S. Ritter, J. R. Anderson, K. R. Koedinger, and A. Corbett, "Cognitive tutor: Applied research in mathematics education," *Psychonomic bulletin & review*, vol. 14, no. 2, pp. 249–255, 2007.

[15] T. Zhang, G. Upadhyaya, A. Reinhardt, H. Rajan, and M. Kim, "Are code examples on an online q&a forum reliable?: A study of api misuse on stack overflow," in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, IEEE, 2018, pp. 886–896.

[16] S. Zhu, Z. Zhang, B. Qin, A. Xiong, and L. Song, "Learning and programming challenges of rust: A mixed-methods study," in *Proceedings of the 44th International Conference on Software Engineering*, 2022.

[17] A. Zeng and W. Crichton, "Identifying barriers to adoption for rust through online discourse," PLATEAU, 2018.