

**Proceedings Of The
Rust-Edu
Workshop 2022**

20 August 2022



Ed. Bart Massey

Rust-Edu Workshop 2022

20 August 2022

The 2022 Rust-Edu Workshop was an experiment. We wanted to gather together as many thought leaders we could attract in the area of Rust education, with an emphasis on academic-facing ideas. We hoped that productive discussions and future collaborations would result.

Given the quick preparation and the difficulties of an international remote event, I am very happy to report a grand success. We had more than 27 participants from timezones around the globe. We had eight talks, four refereed papers and statements from 15 participants. Everyone seemed to have a good time, and I can say that I learned a ton.

These proceedings are loosely organized: they represent a mere compilation of the excellent submitted work. I hope you'll find this material as pleasant and useful as I have.

Many thanks to the Program Committee — Cyrus Omar and Will Crichton. They put a lot of work into organizing and reviewing to get this thing going. Thanks to Futurewei, Inc for providing the funding to make this all possible. Thanks also to the other volunteers who helped make the event a success, and to the contributors and participants.

Bart Massey
1 September 2022

(Cover art by Bart Massey from CC-0 assets, slightly retouched using Stable Diffusion.)

Copyright © 2022 The Rust-Edu Organization

All materials contained herein are copyright their respective authors and creators, and are used by permission.

Table Of Contents

- Statements — p. 4
- Papers — p. 27
- Slides — p. 59

Rust For Students, By Students

Forest Anderson (*forestanderson@cmail.carleton.ca*)

Rust Education Workshop 202

This summer, I taught students at Carleton University the Rust language in a student-led course titled "Summer of Rust". It was not an official course, but ran more in the capacity of a student club or society. Around 40 students registered for the course, and around 20 stuck around past the halfway point. The format was 1.5 hour lectures, and three options of sections at set times each week.

At the start of the summer, I took some time to think about what value this course would have for students. I knew that it could be just a fun thing for them to do, but I also wanted to answer the question of "why Rust?". There are many languages that could help students in different ways, for many different problem statements. However, I came up with some core areas that defined what I wanted students to get out of the summer:

Rust's constraints have well-explained solutions

Although several of Rust's concepts are unique and novel to students familiar with Java-like languages, Rust's learning curve is significantly more approachable through the guidance of the compiler.

The multi-paradigm nature of Rust

Approaching topics that have been hurdles in the past for students is made easier in Rust through semantics and language constructs. Some examples of this include the rules of mutability, the lack of null references, or iterators for data manipulation.

Exploration past what school teaches

Through my education in our computer science degree, I feel like many courses scratch the surface of topics such as parallel computing or functional programming, but don't go far enough to excite the students with what they could do on their own. A large reason for this is the overhead of exploration without feeling too stuck with complex toolchains.

Moving forward, I want to run another iteration of this course in the fall semester. I particularly want to improve on the interactive portion of each lecture, as well as how students can work together more for their labs.

Universidade Federal do Ceará
Campus de Quixadá

Aug. 16, 2022

Rust-Edu Workshop 2022

Rust is nice for making us think before we code

The most delightful feature I see with the Rust language is its ability to codify application requirements with its type system and ownership semantics. This, along its choice of being semantically explicit, allows us to create very pleasing to use APIs. This is a very nice thing to teach to our students.

My concerns are how we prepare our students to better understand and work with the borrow checker and all the syntax and constructs needed to satisfy it, and how we can structure the learning materials so we mitigate Rust's tendency to mix a variety of concepts to solve some problems.

Should we use increasing subsets of the language? Should we hide complexity with a library until they are ready for it? Should we identify a convenient problem that can be iterated upon?

I struggle to see a learning path that can accommodate both students learning to program and those who know their way and just want to learn the new concepts. I hope this community will succeed in this effort and hope I can contribute to it.

Prof. Arthur Rodrigues Araruna
<araruna@ufc.br>

Can't Spell "Curriculum" Without "C"

Rust Education Workshop 2022

Tiemoko Ballo, tballo@alumni.cmu.edu

Alex James, ajames@alumni.cmu.edu

Let's be frank: we don't yet know if Rust is the future of high-performance software. Rust offers a commercially-viable solution to the dire and enduring problem of memory safety without garbage collection latency [1]. Coupled with a growing ecosystem and a long-held "most-loved" status [2], it's a solution with undeniable appeal. But will Rust have *anywhere near* the societal relevance of the C language, even 10 years from now? Is it truly the most appropriate way to teach systems programming to burgeoning engineers with little prior?

The staples – operating systems, computer networks, and embedded development – can readily be taught in Rust. Thus it may be tempting to de-prioritize C within a university curriculum. Or remove coverage outright.

Side-lining C would be a grave mistake: a disservice to today's learners and, by extension, tomorrow's practitioners. From the perspective of a realist, not an apologist. C has formidable pedagogical and pragmatic value. Especially if taught alongside a more modern alternative.

Today's Learners: Type System Contrast

As learners, students should have the opportunity to think deeply about the impact of abstraction design on a problem domain. On one hand, C's simplicity, in count of concepts and keywords, makes it an ideal vehicle for teaching fundamental low-level abstractions. On the other, a relative lack of high-level abstractions (no traits or classes, generics, sum types, references, iterators, collections, async) forces a ruthlessly tactical approach to problem solving. Students must carefully consider mechanical minutia before implementing business logic. Pointer arithmetic/null-state [3], memory allocation, data layout/initialization, undefined behavior, etc. Even the most rudimentary task is fraught with subtle defect potential.

When a learner inevitably introduces a runtime error, they are asked to reason about the many complexities of program execution. Even if C's constructs no longer map directly to hardware [4], a student must develop a reasonably accurate mental model of "what the machine is doing" to fix a segmentation fault. Programming in C remains a literal crash course in computer architecture. Of a visceral nature Rust can't quite imitate.

Now the lasting lesson, the career-long imprint, isn't about the machinery of stack and heap memory. It's about the difficulty of reliable defect elimination in weakly-typed programs. A first-hand taste of C debugging prepares students to appreciate the intent and comprehend the benefit of Rust's borrow checker. Learning both languages elucidates pass-by-reference semantics, with their myriad performance and security implications, via juxtaposition.

That security piece – the connection between an easy-to-introduce bug and a possible-to-exploit vulnerability – is key. In 2021, 67% of zero-day exploits "detected and disclosed as used in-the-wild" relied on memory corruption [5]. Contemporary education must reflect this reality. Through the lens of C, students can understand spatial (e.g. buffer overflow) and temporal (e.g. use-after-free) bug classes. In sufficient depth to diagnose root cause – not develop exploits. And this hard-won knowledge is transferable: professional Rust developers make judicious use of `unsafe` [6], where C-like concepts and risks still apply. Rust's type system is progress, not panacea.

Tomorrow's Practitioners: CFFI Competence

As practitioners, students will go on to build production systems of ambitious scale and complexity. That often means contributing to multi-lingual code bases, where each business problem is solved by the best tool for the job.

Several mechanisms enable cross-language interoperability. In the cloud, modern microservices use standardized response-request formats – like REST and gRPC. But on the client, the C Foreign Function Interface (CFFI) is king. For better or worse, C-style data representation remains *the* de facto protocol for connecting languages [7].

Why? Whereas package managers let us compose source code libraries, all programs ultimately compose at the compiled shared library and process/OS boundaries. Because every mainstream OS is written in C, CFFI powers abstractions at these intersections – like Python's extension modules, the Java Native Interface (JNI), and most application-specific embedded scripting. No matter what level of the technology stack graduates go on to specialize in, they'll never truly escape C semantics. In fact, those who master them are well-equipped to integrate otherwise disparate technologies and deliver impactful solutions. So let's prepare them accordingly.

Closing

Framing Rust in the context of C makes for an accessible introduction to type systems and practical vocational training. Simultaneously. That's why we chose this approach in *High Assurance Rust* – a free online textbook about developing secure and robust systems software [8]. Now there are countless pedagogical challenges in seamlessly bridging a multi-decade (1972 to 2010) language paradigm gap – we need a range of books, courses, and tools as diverse as the learners they serve. Only one thing is certain: the ideal curriculum is largely undefined.

References

- [1] L. Szekeres, M. Payer, T. Wei, and D. Song, “SoK: Eternal War in Memory,” in *2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA, May 19-22, 2013*, pp. 48–62, IEEE Computer Society, 2013.
- [2] “Stack Overflow Developer Survey 2022: Most loved, dreaded, and wanted.” <https://survey.stackoverflow.co/2022/#technology-most-loved-dreaded-and-wanted>, Jun 2022.
- [3] T. Hoare, “Null References: The Billion Dollar Mistake.” <https://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare/>.
- [4] D. Chisnall, “C Is Not a Low-Level Language: Your Computer is Not a Fast PDP-11..” *Queue*, vol. 16, p. 18–30, apr 2018.
- [5] M. Stone, “The More You Know, The More You Know You Don’t Know.” <https://googleprojectzero.blogspot.com/2022/04/the-more-you-know-more-you-know-you.html>, Apr 2022.
- [6] V. Astrauskas, C. Matheja, F. Poli, P. Müller, and A. J. Summers, “How Do Programmers Use Unsafe Rust?,” *Proc. ACM Program. Lang.*, vol. 4, nov 2020.
- [7] A. Beingessner, “C isn’t a programming language anymore.” <https://gankra.github.io/blah/c-isnt-a-language/>, Mar 2022.
- [8] T. Ballo, M. Ballo, and A. James, “High Assurance Rust: Developing Secure and Robust Software.” <https://highassurance.rs>, 2022.

Rust Education Workshop 2022 statement

Jim Blandy jimb@red-bean.com (<mailto:jimb@red-bean.com>)

I am a software engineer at Mozilla, using Rust for my daily work, and the author, with Jason Orendorff and Nora Tindall, of the book *Programming Rust*, published by O'Reilly.

I've been programming professionally since 1990, almost all of that in C and C++. Since 2008, I've been contributing to the Firefox web browser. Shipping C/C++ code to millions of non-technical users, as Mozilla and many other organizations do, is a harrowing responsibility, once you come to terms with what's going on: a small mistake in your reasoning can create a security vulnerability, and malicious actors are constantly evaluating our code to look for such mistakes. This places a crushing burden on the developer and reviewer to protect the users by, essentially, being perfect. In that context, a memory-safe systems programming language like Rust looks like a godsend.

I would very much like to see Rust taught at the undergraduate level:

- **It's worth teaching systems programming to undergraduates.** By 'systems programming', I mean programming whose consumption of system resources like memory and processor time is a primary concern. Undergraduates benefit enormously from gaining "mechanical sympathy" for the machines they'll be using in their careers.
- **Rust is good for teaching.** Its memory safety, thread safety, and excellent diagnostics make debugging much easier, reducing frustration and distraction for students. (Of course, the rules that ensure this safety may be difficult to teach.)
- **Rust is good for teaching systems programming.** Unlike any other memory-safe programming language, Rust achieves safety without concealing the character of the machine. Unsafe code is available for reaching outside what the compiler can check, but safety is the default.

Rust as a Lens

Rust Education Workshop

Chris Johnson (johns8cr@jmu.edu)

For the upcoming semester at my teaching-oriented university, I am reworking our programming languages course to examine language concepts and design decisions through the lens of three unique but non-niche languages. Despite having very little direct experience with the language, I have chosen Rust to be one of the three. We will visit Rust last and contrast its take on modularity, code reuse, lambdas, and pattern matching with the languages we will have discussed earlier. Additionally, we will use Rust to start new discussions on memory management and concurrency.

I have made half-hearted attempts to learn Rust in the past year. I intended to use it for the Advent of Code challenge, but I frequently jumped ship to languages in which I would be more productive. I read blog posts and a couple of books, but I wasn't writing any Rust to reinforce what I was learning. This summer my learning is finally getting grounded as I write an interpreter for a markup language, though I stumble from borrow error to borrow error. Recently I discovered Exercism and have been working through its Rust exercises. Contrasting my solutions to others has been very informative. Perhaps I have chosen Rust as one of our lenses because teaching something is the best way to learn it. My poor students.

My interest in Rust stems from my desire to find a statically-typed language that I enjoy using. I spent many years writing C++ code and felt competent in it. But the language got uglier and uglier, and I felt less inclined to keep up with its new features. I now have no interest in writing any more C++ code in my life. My first decade of teaching was centered on teaching programming courses in Java, but my interest in it eroded for two reasons. First, I encountered lambdas and pattern matching in other languages and fell in love with their brevity. Java has those features now, but they arrived too late. Second, Java took on a corporate feel after its change of ownership. I don't want to use a language that's at the center of lawsuits.

I don't know if Rust is the language that will fill the statically-typed void in my life. However, it has a number of positive qualities. I find myself thankful for its speedy compilation and execution. Thoughtful people have written excellent books and documentation for it. An open community is advancing the language. It doesn't feel beholden to decisions made under constraints from 30 years ago.

Like a piano, Rust expects me to learn how it works in order to use it properly. I feel some ambivalence about this learning curve. On the one hand, I feel respected as a developer when a tool doesn't patronize me and become more toy than tool. On the other hand, I have the responsibility of teaching this language to students in just a few weeks in my programming languages course. I'm eager to explore this tension in this workshop.

Rust Education and Rust Advocacy in India

Rust Education Workshop 2022

Rohit Dandamudi <rohitdandamudi.1100@gmail.com> <https://diru.tech/>

(Open Source Software Engineer, Associate Fellow at Rust Foundation)

I am Rohit Dandamudi, a Rustacean from India. I think the lack of awareness with respect to Rust in India is causing a loss of opportunity to make this language reach more people. The preconceived notion of having a high learning curve and the idea of getting started with a systems programming language, unknown use-cases/people to look up to, etc is acting as a barrier to entry to an awesome experience and community..

Engineering in India is one of the most sought-after courses and is getting adapted every day with a plan to inculcate practical and modern technologies in its curriculum. My goal is to get some traction to bring the community together and create interest so that people from any level can come in and create a way to build software, do research, and maybe find the field that they are passionate about through Rust i.e making Rust be of the real options to learn/have a career in India and the rest of the world.

The Rust Foundation has enabled me to work on Rust advocacy in India, and as part of that, I am trying to introduce Rust in an accessible way to college students as well as anyone who wants to get started. In an effort toward that, I have reached out to a few local institutions that have already shown interest to add Rust in value-added/extracredit courses. Attending this workshop and further collaboration with the rust-edu community will help me learn from/discuss with others and put forth a more formal and structured proposal that will be seriously considered.

I believe there is a lot of unrealized potential and I want to be part of making Rust-based education reach more people and create a new era in Computer Science and Engineering.

Thank you for your efforts in creating this platform!

Rust as an Intersectional Language (with Baggage from History)

Rust Education Workshop 2022

Galileo Daras <galileo@getcoffee.io / galileo@worldcoin.org>

Emphasizing Rust's placement at the intersection of computer science and software engineering could serve as the foundation for a radically different approach to teaching computer science as a whole.

Rust, by enabling the application of formal logic to deep systems programming applications, has gone back to address a divide that has been deepening for the last thirty plus years. Academic computer science largely collapsed on the usage of declarative/functional/verifiable programming languages in large part because those languages were purpose built for that setting. By contrast, practice software engineering collapsed largely on imperative/applied languages because they were used to ship important pieces of software rather than for the features of the languages themselves. This rift has, anecdotally, been poorly addressed or even ignored in computer science curriculums.

There have rarely been classes for computer science which have endeavored to offer a solid and pragmatic foundation when compared to other sciences. Introductory physics courses, for instance, build up to our modern-day understanding of the world by analyzing the evolution of thought and looking at where mistakes were made and what we are building on top of today. Leveraging the history-first approach to teaching the sciences, a first-of-its-kind curriculum could be developed that looks at the history of ALGOL, C, ML, and more to explain the significance of Rust and the origin of many modern programming shortcomings.

In summary, my hopes for Rust education are:

- An emphasis on teaching Rust as the intersection of computer science and software engineering, not a memory-safe C++ alternative
- Leveraging the learnings from other science-focused curriculums to teach the modern state of the programming by detailing the turbulence in its history

Why should we learn Rust?

Rust Workshop 2022

Nisha P Kurur

August 19, 2022

I work as a coding skills instructor at ASAP Kerala, India. Recently, we had an evaluation of the languages to be included as part of the coding skills programme. That is when I realized the importance and popularity of Rust language. The declaration of Rust merging into Linux kernel this June, shows how fast this has been growing in the systems programming world. In my opinion, Rust should be taught along with C/C++ so that the advantages of this language can be genuinely appreciated by the learners. Programmers will never have to worry about Segmentation faults and Code dumps :)

Python and Rust are equally growing these days. Python was introduced in 1991 and it took a long time to get popular. However, Rust was introduced in 2010 only and has a steady increase in its user base within the last decade. This is mainly because Rust makes sure that there are no circumstances where the program can get into uncertain memory conditions thereby assuring runtime efficiency and safety.

Rust also comes with excellent support mechanisms with a single tool called Cargo. This tool is capable of compiling code, run tests, generate documentation, create and upload packages to a repository and many more such features that makes it more attractive and easy to use. Due to all the above mentioned features and characteristics, Rust language was selected as the most loved language of 2020.

Eventhough there are many advantages, the main disadvantage of this language is its learning curve. It is not a very easy language like python and needs special attention and focus on grabbing its concepts and reusing it in the appropriate manner. The addition of Rust language to Linux kernel (supposed to be happening by 2023) would positively boost the user base and help in popularising it better. Also, more job opportunities would be created in Rust due to this change in Linux kernel which will further require more training/teaching opportunities for Rust language.

Who learns Rust? Those with orthogonal concerns with shared interests

Rust Education Workshop 2022

Tim McNamara¹ <timmcn@amazon.com>

Statement

Many learners are attracted to the Rust programming language, but few come to the language with the same background. Attempting to create learning materials and training programs that are too broad risks alienating people, rather than helping them to achieve their goals.

96% of respondents to the 2021 Rust community survey (p42) agree that Rust provides a real benefit over other languages, but 62% believe that Rust is harder to learn than other languages. There is a disconnect, with optimism on one side and frustration on the other. It's fixable, yet it's present.

Creating an effective teaching program involves deeply considering the needs of your learners. From that survey, we can see that about 85% of people in the Rust community have more than 5 years' experience (p6). But those experiences are vastly different. Someone who has only programmed in Python may not know what the terms reference or memory safety mean. Her needs are different to someone else who has been a C++ developer for a decade or more.

96% of your learners share an optimistic future. They want to be able to write safe, reliable and resource efficient software. Let's learn how to teach Rust effectively. I suspect that will require us to teach Rust empathetically.

Bibliography

2021 Rust Community Annual Survey <<https://is.gd/1TYhC6>>

¹ Tim McNamara is Senior Software Developer at AWS. He has responsibility for Rust education within Amazon, as part of a global team that has been established to foster the use of the language. As the author of *Rust in Action*, he has assisted hundreds of thousands of programmers with their journey to learn Rust.

Rust-Edu Workshop Statement (August 2022)

Dylan McNamee (dylan.mcnamee@gmail.com)

My suggestion

My suggestion is to introduce Rust as a *second* programming language. I think that for *most* students simpler languages, like Python or even Scratch, are well-suited as first programming languages where they learn concepts of step-by-step execution, flow control, even basic abstraction into functions. For non-CS-destined students, Python might be sufficient as the only actual programming language they learn.

In many curricula after learning the basics of programming using a garbage collected language students are taught a *systems programming* class where memory is more of an issue and is usually managed manually (Go is a notable outlier here). I suggest that Rust be taught at this level for the following reasons:

- Rust is more difficult to achieve basic proficiency at than Python, so I think Python is a more welcoming on-ramp to students who might be scared off by having to tackle memory management in their first language,
- Rust's initial curve is steeper than basic C/C++ (I realize this may be controversial for this audience), however
- Rust is easier to learn than learning C/C++ well enough to not make serious/dangerous programming errors

My plan

I'm going to teach an upper-division class at Reed College this Spring (2023). The topic of the course is "Intro to Operating Systems," but I have publicly subtitled the class "Let's learn Rust and write an Operating System in it." There is a lot of material in an Intro OS course, so adding learning a new language (incoming students will have had one "Systems" class, where they were exposed to C++) is a tall reach, so I will seek to teach the class with a subset of Rust's features. The challenge I have between now and Spring is to figure out the concrete project (from among many available!), the actual subset of Rust we'll be using, and how to introduce that subset as smoothly as possible.

A possibly controversial addenda: data-oriented programming

I'm seriously considering teaching *data-oriented programming* as an alternative paradigm to object-oriented programming. As a practitioner, I have found that creating an object-oriented system that doesn't bog down in class hierarchy complexities, and that most of the goodness of OOP is captured by appropriate use of *interfaces*. In my early experience Rust is a great match for data-oriented programming (though DOP can be used in almost any programming language).

Experience Teaching Compilers Using Rust

Max S. New

maxsnew@umich.edu
University of Michigan CSE

August 18, 2022

In the Fall 2021 semester, I taught EECS 483¹, compiler construction using Rust in two ways: (1) As the language in which students wrote their compilers (from a functional language to x86 assembly) (2) As the language in which students implemented the runtime system for their compilers.

I based the course off of one which had been taught at Northeastern University by Benjamin Lerner²³. Lerner’s version of the course uses OCaml as the compiler implementation language and C for the runtime system. I had concerns about using these languages at the University of Michigan, where most of the core courses taught using C++. My main motivation for using Rust for the course was to be able to follow a similar structure to the OCaml course, using polymorphic functions and datatypes as well as tree types with pattern matching, but using a language more similar to the C++ programming they were used to.

The choice of Rust paid off in several ways. First, the tooling around Rust was of very high quality: installation, testing and cross-compilation support were straightforward for students. In particular, implementing the runtime in unsafe Rust meant the compiler and the compiled code could easily be built with Cargo. Second, students were happy to learn what they considered a “cool”, new language and so this mitigated in some ways the resistance to learning new language features. While lifetime errors were common, the students found in my experience the notion of lifetimes and smart pointers to be natural based on their C++ experience.

The main difficulties with Rust were common ones: infection of codebases with lifetime annotations and troubles with linked list datastructures. In particular, if we used str slices in our syntax tree types, the datatype would need to be parameterized by a lifetime. Lifetime elision usually made mentioning this unnecessary, but then students would often be confused the first time they wrote a function where lifetime elision did *not* work. The second difficulty was that shared linked list datastructures are quite natural in compiler code, for example representing environments of bound variables, but they are not very ergonomic in Rust and are not featured in the standard library. In fact, I initially used vectors before realizing the amount of copying it required.

¹<http://maxsnew.com/teaching/eeecs-483-fa21/>

²<https://courses.ccs.neu.edu/cs4410/>

³The course has also been adapted by Joe Politz, using TypeScript as implementation language and targeting WASM (<https://ucsd-cse231-s22.github.io/>)

RustViz: Interactively Visualizing Ownership and Borrowing

CYRUS OMAR, MARCELO ALMEIDA, GRANT COLE, KE DU, WANGRUI (EMELIA) LEI, YIXIAO (SERENA) LI, GONGMING LUO, SHULIN PAN, YU PAN, KAI QIU, VISHNU REDDY, HAOCHEN ZHANG, ZHE ZHANG, QIYUAN YANG, YINGYING ZHU, Future of Programming Lab, University of Michigan, USA

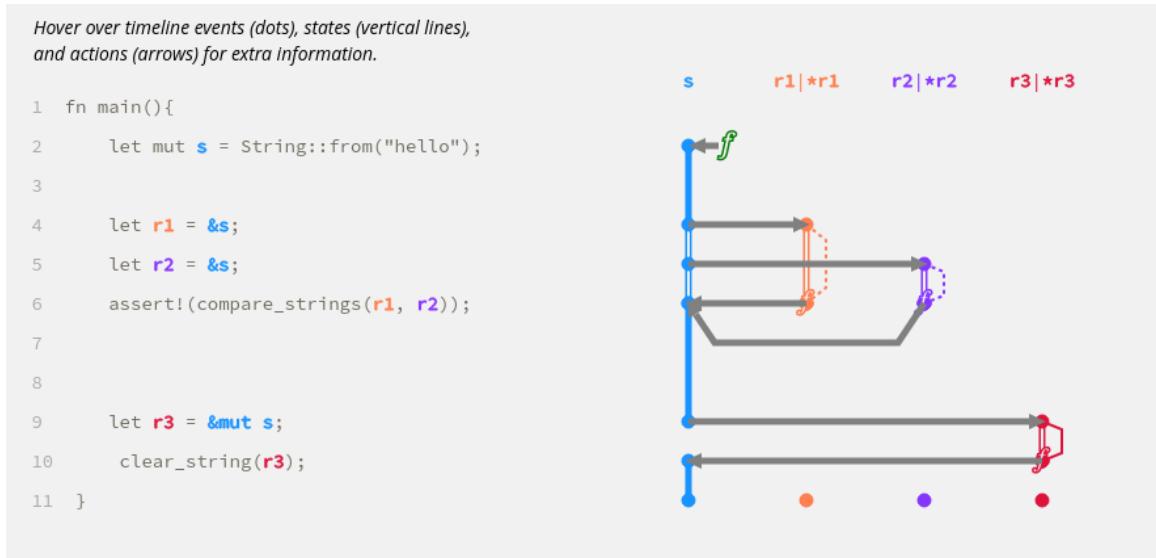


Fig. 1. An example demonstrating various aspects of the RustViz interactive visualization system.

Rust is a systems programming language unique amongst its industrial peers in achieving memory safety without the need for a runtime garbage collector. Instead, Rust relies on a unique and sometimes subtle resource ownership and borrowing system, together with a variety of known-safe library primitives (internally implemented using an escape hatch that permits locally unsafe memory operations). Collectively, this can make learning Rust a challenge for novices and experienced programmers alike [4, 6]. A systematic survey of posts on various online forums found that people learning Rust frequently complained that the borrow checker, whose influence cross-cuts the entire ecosystem, was inaccessible [9]. In interviews, participants learning Rust reported that the borrow checker was an “alien concept” and “the biggest struggle” in learning Rust [8]. Interviews with industry professionals also found that difficulties with learning Rust presented barriers to adoption [7]. Based on the studies above, we identified a key challenge: the programmer must learn to mentally simulate the logic of the borrow checker, which is known to be challenging [2, 3, 5].

To help students learn to understand the borrow checker, we built a tool, RustViz, for creating interactive visualizations that explicitly visualize ownership and borrowing events (Figure 1). Users can hover over each entity in the visualization, e.g. points, lines, and function symbols, to receive a detailed explanation. This visualization can be integrated into instructional material constructed using mdbook, the documentation generator used ubiquitously in the Rust ecosystem (including for the official Rust Book). We developed and evaluated RustViz in the context of a single university course on programming languages. The results have been excellent, as described in a recent publication to appear at VL/HCC 2022 in September [1]. We will introduce RustViz to the workshop audience and argue for its broader use in a variety of Rust-based courses.

REFERENCES

- [1] Marcelo Almedia, Grant Cole, Ke Du, Gongming Luo, Shulin Pan, Yu Pan, Kai Qiu, Vishnu Reddy, Haochen Zhang, Yingying Zhu, and Cyrus Omar. 2022. RustViz: Interactively Visualizing Ownership and Borrowing. *To appear, 2022 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)* (2022).
- [2] José Juan Canas, María Teresa Bajo, and Pilar Gonzalvo. 1994. Mental models and computer programming. *International Journal of Human-Computer Studies* 40, 5 (1994), 795–811.
- [3] John M Carroll and Judith Reitman Olson. 1988. Mental models in human-computer interaction. *Handbook of human-computer interaction* (1988), 45–65.
- [4] Will Crichton. 2020. The Usability of Ownership. In *Proceedings of Human Aspects of Types and Reasoning Assistants (HATRA '20)*.
- [5] Will Crichton, Maneesh Agrawala, and Pat Hanrahan. 2021. The Role of Working Memory in Program Tracing. In *CHI '21: CHI Conference on Human Factors in Computing Systems, Virtual Event / Yokohama, Japan, May 8-13, 2021*. ACM, 56:1–56:13. <https://doi.org/10.1145/3411764.3445257>
- [6] Kasra Ferdowsi. 2022. *The Usability of Advanced Type Systems: Rust as a Case Study*. Retrieved June 9, 2022 from https://weirdmachine.me/papers/usability_of_advanced_type_systems.pdf
- [7] Kelsey R. Fulton, Anna Chan, Daniel Votipka, Michael Hicks, and Michelle L. Mazurek. 2021. Benefits and Drawbacks of Adopting a Secure Programming Language: Rust as a Case Study. In *Seventeenth Symposium on Usable Privacy and Security, SOUPS 2021, August 8-10, 2021*, Sonia Chiasson (Ed.). USENIX Association, 597–616. <https://www.usenix.org/conference/soups2021/presentation/fulton>
- [8] Nischal Shrestha, Colton Botta, Titus Barik, and Chris Parnin. 2020. Here We Go Again: Why Is It Difficult for Developers to Learn Another Programming Language?. In *Proceedings of the 42nd International Conference on Software Engineering, ICSE*.
- [9] Anna Zeng and Will Crichton. 2019. Identifying Barriers to Adoption for Rust through Online Discourse. *CoRR* abs/1901.01001 (2019). arXiv:1901.01001 <http://arxiv.org/abs/1901.01001>

An overview of a Rust university course

By Henk Oordt <hd@oordt.dev>

Introduction

I have been using Rust professionally for about 4 years. Apart from backend development, I'm mainly using Rust for embedded. I love how Rust makes me productive in writing complex applications. I have been expressing my love for Rust by organizing workshop, both internally and for external companies that are interested in using Rust. Making other people enthusiastic for this technology is one of the best things in my job.

In the following months, me and some of my colleagues will be working on a 12-week course for a Europe-based university. We will be organizing the course once, and after that publish the material for reuse by other institutions. After that, the university should be able to keep the course running by itself. The main goals are to give students and future teachers very practical, hands on experience. We want to show them why we like using Rust so much. During some of the workshops, we found out that emphasis on why Rust is useful is key to keeping attention of the participants. This is why we'll be starting with the 'why' of Rust, and mentor students intensively with their exercises during the course. That way, we hope to mitigate any issues with Rust's steep learning curve. Apart from that, we believe the students are equipped best if they put Rust into practice in a bigger project. That is why a relatively large portion of the course focuses on helping students out with the projects they'll propose themselves.

Below I have listed my draft course outline, which is not final or even complete as of this moment. It was more or less copied from my notes. However, it may serve your efforts in setting up a university course. I would like to get involved with Rust-edu in some way, in order to share ideas or even by creating some course material or exercises for the initiative.

Draft course outline

Target audience

- ± 20 students from CS faculty
- ± 3 future teachers
- intermediate level C++ knowledge

High-level goals

1. Ability to write custom CLI/server applications using popular crates or to contribute to existing projects
2. Ability to teach Rust to other people
3. Get practical, hands-on experience
4. Deep dive, intermediate level Rust skill
5. Know the problems Rust aims to solve
6. Ability to judge whether Rust fits project requirements
7. Know why Rust features are the way they are

Low-level goals/topics

0 - Course introduction

- Goals
- Structure
- Schedule
- Project introduction
- Tools
- Contact info

A - Introduction to Rust

A1 - Language basics

- **'Why' of Rust language**
 - Problems Rust intends to solve
 - The fields it operates in
 - Rust design goals
 - Why Rust is considered secure
- **When to use Rust**
 - Where Rust really shines
 - Where Rust maturity lacks
 - What Rust wasn't designed for
- **basic syntax and operators**
 - Types: primitives/struct/enum/union/slice/String/Vec/Box/Option/Result
 - Control flow
 - Scopes, blocks, statics
 - Expressions
 - Functions
 - Pattern matching
 - Loops
 - Impl blocks
 - Coercion
 - Closures
 - Comments
 - Casing conventions
 - ...
- **Structure of a Rust application**
 - imports
 - main function
 - modules
- **Conversions**

- casting/as and pitfalls - `.into()`, `.try_into()` `T::from()`, `T::try_from()`, but not yet the traits they originate from

- **Panicking: explicit/unwrap/overflow**

- What happens on panic
- `no_panic`
- When panicking is OK, and when it's not

Exercises *TBD*

A2 - Ecosystem and tools

- **Cargo**

- configuration
- dependencies
- cross-compilation
- `rustup`
- `rustfmt`

- **Build profiles**

- debug vs. release
- Opt-levels
- LTO
- ...

- **Tour through crate index and API docs**

- [Docs.rs](#)
- [Lib.rs](#) (unofficial)
- [Crates.io](#)

- **Widely used tools**

- debug
- test
- security
- bench (Criterion)

- **Rust Nightly**

- Release cycle
- Unstable features

- **More resources:**

- TRPL
- [cheats.rs](#)
- reference
- rustonomicon

Exercises *TBD*

A3 - Ownership, references

- Pointers vs references, reference meta
- Copy, clone, moves
- Ownership, borrowing, lifetimes
- Lifetime annotation, elision
 - Why needed
 - Syntax
- Helpful types
 - Cell/RefCell/Rc

Exercises *TBD*

A4 - Traits and generics

- Traits
- Commonly used traits from std
 - Into/From/TryFrom/TryInto
 - Copy/Clone
 - Debug/Display
 - Iterator/Intolter/Fromlter
 - FromString
 - AsRef/AsMut
 - Deref/DerefMut
 - PartialEq/Eq/Add/Mul/Div/Sub/PartialOrd/Ord
 - Drop
- Generics, trait objects, object safety, const generics
- Orphan rule
- Macros

Exercises *TBD*

B - Application programming

- Structure of a Rust project
- Setting up a Rust crate, bin vs lib
- Error handling: enum/anyhow>thiserror
- [Rust API guidelines](#)
- Widely used crates: logging/argparse/(de)serialization/testing
- Build scripts
- Conditional compilation, features
- Improving build time

Exercises *TBD*

C - Multitasking

- Atomic types
- Multithreading: Send/Sync/Channel....
- How the borrow checker helps us
- Smart pointers, std::sync
- Async: mechanics/Future/pin/runtimes
- Tokio runtime deep dive

Exercises *TBD*

D - Idiomatic Rust patterns

- Newtype
- Typestate
- Builder
- Composition over inheritance
- Anti patterns
- ... **Exercises** *TBD*

E - Rust for web

- Rust web crates
 - Hyper
 - Rocket
 - ORM
 - ...
- std::net

Exercises *TBD*

F - Unsafe Rust

- Why safe vs unsafe
- Undefined behavior
- Unsafe keyword
- Added functionality
- Abstract machine
- Optimization
- MaybeUninit
- Drop check, ManuallyDrop
- Type memory layout
- MIRI

Exercises *TBD*

G - FFI and Dynamic modules

- FFI in Rust, extern "C"
- sys-crates
- std::ffi
- catch_unwind
- bindgen
- Cxx/PyO3
- Panics and catch_unwind
- libloading crate
- WASI with wasmtime

Exercises *TBD*

H - Optional

- Rust on embedded
-

•

P Final project

Ideas

- Scientific programming: nalgebra
- Game development: ggez, bevy, <https://www.arewegameyet.com/>
- GUI application: <https://www.arewoguiyet.com/>
- Doubly linked list
- Embedded programming
- Some audio filtering and streaming software
- Contribute to an open source project
- SIMD

Structure

- Work in teams of 2
- Hand in proposal in week 7
- Write small report (2-3 pages), to be handed in in week 12
 - Introduction
 - Requirements
 - Design diagram
 - Design choices
 - Dependencies
 - Evaluation
- Present project in about 5 minutes in week 12

Course structure

- 7 modules
- 12 weeks, 1 lecture of 90 minutes and 1 tutorial of 90 minutes per week
- 1 exercise set per lecture, to be completed during tutorial or at home
- 1 final project with small report to be completed during mentoring session or at home

Schedule

Week	Date	Module	Lecturer	Notes
1		0, A1		Course intro
2		A2, A3		
3		A4		
4		B		
5		C		
6		D		Project proposal reminder
7		E		Deadline project proposal
8		F		Project proposal resubmission
9		G		Start final project
10		P		
11		P		
12		P		Final project submission and presentation

Experiences of Teaching Rust and Code Recommendation to Assist Rust Beginners

Hui Xu
School of Computer Science
Fudan University
<xuh@fudan.edu.cn>

Rust-Edu Workshop 2022
<https://rust-edu.org/workshop>

I am a researcher in the field of software reliability and have been engaged with Rust since late 2019. While I was working on a research project related to Rust, I was attracted by the language and realized its huge potential. So I decided to teach Rust in my class at Fudan University. Luckily, I was able to start a new post-graduate course named Memory Safety and Programming Language Design in the semester of 2022 Spring, which mainly discusses the design intention and underlying mechanism of Rust. In this talk, I will first share my experiences and thoughts about teaching this course, and then present our ongoing research projects that aim at assisting Rust beginners in improving the quality of their code.

While there are many discussions on social media (e.g., Reddit and Quora) complaining about the steep learning curve of Rust, I think Rust is difficult to learn mostly because of two reasons. The first reason is that Rust contains some advanced programming language concepts that are frequently used in real-world Rust projects, such as trait and closure. While these features bring many benefits to writing compact and efficient code, they also increase the difficulty of code comprehension and design. Secondly, even though we may restrict Rust to a minimum subset of features for beginners, it still involves enforced exclusive mutability and lifetime check that bring obstacles to writing compilable code. Rust beginners may suffer from these painful points and get frustrated.

To smooth the learning curve of Rust and help Rust beginners to write decent code, we are currently working on several novel features based on the Language Server Protocol or Rust Analyzer. We think the protocol offers a promising framework because it enables us to leverage the server-side computing power to perform code analysis periodically and to make suggestions to developers. Therefore, we can do more complicated code analysis jobs than the current Clippy or lint can do. For example, we may detect replaceable unsafe code leveraging machine learning techniques and recommend equivalent safe versions to developers.

The Problems of Teaching Rust in Academia

Yury Zhauniarovich
y.zhauniarovich@tudelft.nl

August 18, 2022

I have been interested in Rust for several years. The first time I heard about this language was before its 1.0 release. At the time, I read several opinions about this language and was impressed by the promises that the language provided. I started to follow the updates of this language and decided to give the first time to learn it after the release of 1.0 version. At the time, just a few sources were available, and I decided to use the official Rust Book [1] to learn it. I passed through the book, reproducing the examples available there. Of course, not all the topics were completely clear to me, but I know that doing practical exercises can improve the understanding of the intrinsics of the language. Besides that, at the time, I also developed a deck of Anki¹ cards using the content of the book and used them pretty often. Unfortunately, I did not start programming using this language; thus, with the lapse of time, I forgot what I had learned. The second time I attempted to learn Rust using another book called “Programming Rust: Fast, Safe Systems Development” by Jim Blandy et al. [2]. From my perspective, this is a very good book to be used as a course book because it explains a lot of intrinsics of the Rust language, e.g., memory layouts of main data structures. Still, after reading this book, I also had a shortage of time to start using Rust in everyday development, and after some time, I forgot the knowledge obtained.

With all these examples from personal experience, I would like to show that the only way to learn Rust is to start using it actively in everyday development. Unfortunately, in academia, you do not have a lot of luxury to do this. Indeed, in research, we rely a lot on the hands of MSc and Phd students, who have to finish their research in a limited period. This creates a huge pressure on them, and they are not ready to invest a lot of time learning Rust (Rust is known for its steep learning curve). Instead, they prefer developing prototypes in Python, thus, a lot of libraries are written using this language. This creates even more pressure on fresh students to choose Python because the main libraries that you use in your research prototypes are in Python. Thus, professors working in academia do not have a lot of incentives to use Rust.

Thus, even if you want to include Rust in developing some research prototypes, it is hard to do this. As a result, your knowledge in this language starts to degrade if you do not use it regularly. Now, if you need to teach Rust, you do not have good in-depth knowledge to make the students really interested in it, because you cannot answer some questions or your explanations of the language are not very deep. Moreover, this forces professors to create courses covering only the syntax of the language and paying less attention to the intrinsics and advances.

Therefore, I really welcome the RustEdu initiative. It should allow us, academic enthusiasts, to join forces, develop deep-dive materials covering difficult topics (e.g., I have an article on closures [3] that was featured in “This Week in Rust”), share them, and improve them.

References

- [1] S. Klabnik, C. Nichols, and The Rust Community, *The Rust Programming Language*. [Online]. Available: <https://doc.rust-lang.org/book/>.
- [2] J. Blandy, J. Orendorff, and L. F. S. Tindall, *Programming Rust: Fast, Safe Systems Development*, 2nd ed. O'Reilly Media, 2021.
- [3] Y. Zhauniarovich, *Closures in rust*, 2021. [Online]. Available: <https://zhauniarovich.com/post/2020/2020-12-closures-in-rust/>.

¹<https://apps.ankiweb.net/>

Imagining Introductory Rust

Rose Bohrer

Computer Science Dept.

Worcester Polytechnic Institute

<rbohrer@wpi.edu>

Rust-Edu Workshop 2022

<https://rust-edu.org/workshop>

Abstract

Though the Rust community has expressed interest in designing introductory-level computer science courses around Rust, such courses remain hypothetical as of this writing. Thus, this paper has a speculative style, imagining one design for a future introductory-level Rust course. I do so in the context of a discussion on the state of programming languages in introductory courses and a review of students' educational needs in these courses. My goal is to inform the development of such future courses.

1 Introduction

Introductory computer science (CS) courses are at the heart of CS education. For many students, they are the first or only exposure to CS, informing the choice whether to continue with CS. For instructors, they are our largest opportunity to show students which skills are valued in CS.

An introductory-level Rust course becomes an increasingly reachable goal as the language matures. The lack of an introductory Rust course is not for lack of educators in the Rust community. Rather, introductory courses' outsized enrollments and impacts motivate a slow, careful rate of change. An introductory Rust course deserves robust discourse before implementation. To that end, I use my first-person educational experience (consistent with the tradition of Burkhardt and Schoenfeld [3]), to speculate a design for a future Rust course. Many of my design choices may be common sense to readers, but nonetheless remain key to developing robust discourse.

Why Am I Interested in Introductory Rust? The author has several years' experience teaching introductory-level courses in functional programming languages, first teaching CMU's course 15-150 in Standard ML, now WPI's course CS 1101 in Racket. Standard ML and Racket, like Rust, could be considered *minority programming languages*, i.e., their user bases are smaller than Rust's [27]. Instructors of such courses are often asked to justify the use of minority languages, given that large user bases are associated with career potential, a source of student motivation [10], and availability of community support.

In exploring "Why Racket?", I find myself asking "Why not Rust?" If taught successfully, Rust has the potential to expose students to foundational issues ranging from data layout to type systems while side-stepping criticisms of minority languages' syntactic difficulty, lack of applications, and lack of career potential. Because sense of belonging is known to promote underrepresented students' persistence [12] and seeing successful people similar to oneself promotes self-efficacy [2], Rust's intentional inclusion efforts (e.g., its code of conduct [26]) are also attractive.

2 What's Valued in an Introductory Language?

I identify guiding values for course design by summarizing literature on student motivation and personal experiences with instructor constraints. In Section 4, I assess how the proposed course satisfy or fail to embody these values.

Satisfying Student Motivations Available data mostly assess CS as a whole, not language choice of introductory courses in particular. I summarize published data from Carleton [13] and the Data Buddies report

at WPI [10]. Both reported high intrinsic motivation to learn core CS concepts across demographic groups. The WPI data [10] cite career and earning potential as major motivators, with somewhat elevated impact for underrepresented minority students. The WPI data [10] cite social impact as a major motivator, with somewhat elevated impact for women.

Instructors should connect the use of Rust to these motivations, framing it as connected to CS foundations, with potential growth in industrial use and social impact. Beyond mere marketing, this messaging is supportive of student engagement and thus learning.

Instructor Motivations The RustEdu audience, like the author, may be intrinsically motivated to use Rust. Outside this audience, extrinsic instructor motivations such as saving time should also be considered. Auto-grading is a major time-saving approach in introductory education [29], so test synthesis tools like SyRust [25] could be explored to generate tests automatically. Instructors using auto-graders should be aware of attendant meta-cognitive challenges [20] and consider feedback DSLs [18] as one approach to overcoming them.

3 What Would a Rust Course Look Like?

I propose a Rust counterpart to WPI's intro CS course CS 1101. CS 1101 is a 7-week course based on the textbook How to Design Programs (HtDP) [7]. I base my proposal on a recent iteration from the third term of the 2021–2022 academic year. CS 1101 combines lectures where the instructor demonstrates Racket programming with lab sections where students program Racket under teaching assistants' supervision.

CS 1101's explicit learning goals are writing and testing (Racket) programs with lists, trees, user-defined data, recursion, and mutation. Viewed from such a high level, Racket appears as well-suited as Rust. In exploring a Rust version of the course, however, we will observe opportunities to incorporate additional objectives from a typical 4-year CS curriculum [1]. In particular, we add the goal of providing preliminary expose to key systems programming issues such as aliasing and data layout, as well as programming language theory issues of static typing and its correctness benefits.

Assessments include daily active learning quizzes and 3 exams. Moreover, CS 1101's 7 weeks correspond to 7 assignments on the following topics: 1. Function composition 2. Structure definitions 3. Sums and lists 4. Binary search trees 5. Higher-order functions, and 6. Mutation and accumulators. I enumerate how each assignment, in turn, could be redeveloped with Rust at its center.

1. The coding section has students write a one-line program that displays an image on the screen. Graphical programs are a widely-used teaching technique to provide an immediate visual payoff:[19]. I propose keeping the graphical elements in the hope that immediate visual payoffs could stimulate student persistence. Nascent GUI libraries [15] could be used. The written section has students evaluate programs step-by-step. Instructors should distill formal semantics like Oxide [28] into an informal semantics for students to show program evaluation and state change step-by-step. The Rust debugger for VSCode should be used to self-check results. Instruction should emphasize the importance of both state and reduction of expressions to values.
2. Students define product types such dates and metadata for books. This translates directly to Rust. Class time can be spent discussing in-memory layout of `struct` fields, in preparation for systems programming. This assignment introduces the HtDP [7] methodology's requirement of writing type signatures on all functions despite the use of an untyped language. To maximize motivation, types in Rust should be framed as helping students achieve their own goal of bug-free code instead of being an arbitrary requirement. Students use class time to write buggy expressions and see Rust catch bugs. This reinforces the learning goal that static types prevent classes of bugs.
3. The Racket assignment simulates a sum “type” for gifts (flowers, plushies, candy) by defining products for each and testing types at runtime. A Rust version can improve by defining the sum type directly. Students should spend class time writing ill-typed sum expressions and inexhaustive cases to learn about type-checking for sums. This hands-on activity should emphasize that Rust's typechecker helps them meet

their goals by automatically catching certain bugs early. Students should also practice the data layout for tagged sum types, reinforcing the distinction between static typing and runtime tags.

4. The Racket assignment introduces the first inductive data structure, lists, with simple list-processing functions. The Rust version can use box-and-pointer diagrams to teach students about indirection. Constant and mutable references will be introduced. In class, students write cyclic lists and use-after-free bugs on lists, then learn why these programs do not typecheck. This reinforces the learning goal of providing preliminary exposure to core systems concepts.
5. The Racket assignment introduces binary search trees (BSTs). The Rust assignment can be expanded to include binary trees without the BST invariant. Then, students can be shown box-and-pointer diagrams for trees with extreme aliasing (e.g., a linear-space representation of self-similar tree of exponential size, displayed in Figure 1 on page 3). This reinforces the goal of preliminary exposure to systems concepts. Students should step through seemingly-correct code for aliased trees on paper, see incorrect results, and

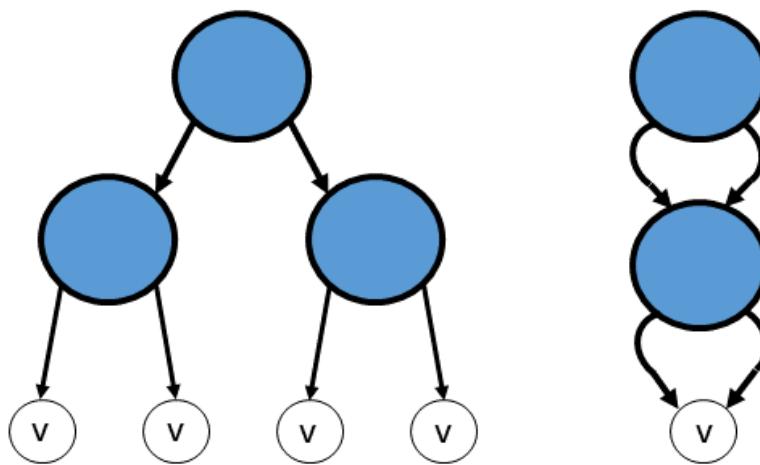


Figure 1: Tree-shaped data structure, both with aliasing (left) and without aliasing (right).

learn affine types prevent creating the aliased tree. Students will write functions with multiple recursive calls on different subtrees, and observe how affine types help prevent recursion blunders (e.g., applying both recursive calls of a destructive function on the same subtree).

6. The Racket assignment practices the higher-order functions (HOFs) `map` and `filter` on lists using a dataset of American rivers. To reduce Amerocentricity, a new version can expand the data set to non-American rivers. All HOFs in the assignment are non-destructive. For simplicity, they should remain so. Class time will be used to discuss the subtleties of destructive HOFs, and in particular, closures that can be used only once. The runtime representation of closures will not be treated as core material, but will be explained in supplementary material. The Racket assignment uses sentinel return values for list-searching functions in the not-found case. Option types will be introduced as an alternative, the *billion-dollar mistake* [11] of null references will be described, and students will discuss option types. This supports the goal of preliminary exposure to big ideas in programming language theory.
7. The Racket assignment models an idealized email system, where messages are sent between users by moving them between lists representing mailboxes. The Rust version will use physical mail as an introductory metaphor for ownership-checking: if I put a letter in someone else's mailbox, I no longer have it. Like many objects, letters can be deep-copied, but copying a physical letter is time-intensive, just as deep copies are compute-intensive. In-class discussion will cover privacy and types: is it possible to ensure that only the intended recipient reads the mail?

4 Opportunities for Success and Failure

This section discusses major points of departure between the Racket and Rust courses, to identify potential risks and rewards of using Rust.

4.1 Syntax and Types as Friend or Foe

Reported student motivation for learning core CS concepts at WPI is high [10]. As a corollary, instructors should be careful with language features that could become viewed as obstacles to the core concepts. In the author's anecdotal experience, students often report that Racket's Lisp-style usage of parentheses confuses them, specifically that they are unsure how many pairs of parentheses each expression should have and where each closing parenthesis should appear. In prior experience teaching Standard ML-based courses, students reported similar issues with parentheses and the common appearance of type errors whose purpose was not made clear to them.

In contrast to Lisp-style parentheses syntax, Rust emphasizes an infix syntax which significantly overlaps with the most common procedural languages. This is potentially helpful in reducing syntactic because 83% of first-year WPI CS students have some prior programming experience [10], but its efficacy should be tested.

The teaching of types requires additional care. To maintain a positive classroom attitude, I cast well-typed programs as rewards instead of casting type-errors as punishments. Firstly, the coursework is designed to highlight specific classes of common errors which static types rule out. This message is reinforced by having students hand-write and hand-evaluate buggy, ill-typed programs, rather than asking students to trust that the type-checker improved their code. Having students intentionally write ill-typed code also normalizes type errors as a standard step in the development process, making clear that they are not a personal failing. Special attention must be paid to the readability of type error messages. To avoid bias in favor of instructor intuition, readability should be researched with empirical studies of novice Rust programmers. DrRacket's error messages were developed with a similar approach [16, 17], which could be emulated. If classroom use mandates domain-specific messages, the VSCode plugin could be customized for classroom use.

4.2 Mature and Immature Tooling

Students do not interact with languages in the abstract, they interact with languages' tools. Thus, tooling is essential to shaping students' first impressions.

CS 1101 uses DrRacket [8], a specialized IDE for educational Racket. DrRacket installs via a standard graphical installer. CS 1101 emphasizes using the built-in help center to learn the usage of individual functions. The DrRacket help center can be optimized for individual courses, i.e., the student can pick the language fragment they wish to use and then restrict the help to show only relevant functions for their fragment. IDE-provided feedback include display of variables' binding sites, jumping to code from error messages. In addition to breakpoint debugging, the DrRacket stepper displays step-by-step evaluation traces for a pure fragment of the language. DrRacket is a direct descendant of the DrScheme [9] editor for Scheme. Both DrRacket and DrScheme are targeted at educational use.

Installing a full Rust development process is a multi-step process. Visual Studio Code (VSCode) provides a featureful graphical editor, but Rust must be installed separately, as are Rust extensions to VSCode, of which several exist [23, 5]. The `crates` package manager may need to download additional dependencies for each program. A Rust version of CS 1101 should provide a one-step GUI installer that installs Rust, VSCode, a Rust extension for VSCode, and all crates used in the course. This prevents installation from being a barrier to entry.

VSCode's counterpart to a help center is IntelliSense, which supports name-completion, jumping to definitions, error highlighting, and display of documentation. Rust supports IntelliSense. IntelliSense should be taught explicitly and documentation should be provided for all starter code. VSCode supports traditional breakpoint-style debugging, instead of a step-by-step display of operational semantics. Mutable state is crucial in Rust, thus breakpoint debugging may be preferable.

VSCode has recently received media attention [24] as a popular IDE. Students should be encouraged that VSCode is a skill they can use in their careers.

4.3 Relevant and Irrelevant Languages

In the instructor's anecdotal experiences teaching Racket and Standard ML, students frequently express concerns that neither is widespread in industry. Because many WPI CS majors pursue computing careers [10], these concerns are well-founded. I outline potential instructor responses.

1. I could argue that language choice should be picked to optimize learning, not applicability [22]. I do not make this argument because I do not yet have data supporting a given language choice.
2. I could argue that language choice should be picked on usefulness in students' career, because this is motivating [10]. I do not make this argument because none of Rust, Racket, or Standard ML are the world's most popular language, though Rust ranks highest at 22 according to TIOBE [27].
3. Instead, I argue that in the absence of conclusive data, I should choose a language which has potential on both fronts, which helps produce research data to guide future curricula. Rust's potential to teach CS concepts is outlined in Section 3: in particular, its strong type system forces students to engage with the concepts that type system embodies. Its industrial potential is highlighted by user base growth [27].

5 Path to Implementation and Conclusion

Introductory courses are large and often slow-changing, so a complete vision for Introductory Rust must include a path toward implementation, outlined here. The path toward implementation, like the paper overall, is guided largely by my personal experience. I do this because although the literature on introductory programming courses is extensive [4, 6, 19, 21, 14], it has not yet addressed the tradeoffs of using Rust in a course nor the best ways to implement a Rust course specifically.

We should the lack of direct prior work as a challenge, working with CS Education researchers to rigorously study Rust's advantages and disadvantages regarding learning outcomes and student satisfaction, with particular attention to the perspectives of marginalized students.

My own institution is well-positioned for such research, though not uniquely so. My course, like many, has multiple lecture sections. Multi-lecture courses are an ideal place for empirical research. I propose courses where each section uses different languages to enable comparison. Differences among instructors can be compensated by comparing the Rust instructor's student feedback against the same instructor's feedback in previous Racket-based iterations. Because WPI uses a 7-week term, only 7 weeks' material need revision. Likewise, instructors elsewhere should exploit any local opportunities to pilot short-format courses.

Introductory courses deserve serious research even before initial implementation. In this, instructors should acknowledge that students know things we do not. Undergraduate researchers should be used to collect formative feedback from classmates to identify strengths and weaknesses of proposed courses. For those of us whose institutions require student capstone projects, we can advise this research as a capstone.

In conclusion, the choice of introductory language should never be taken lightly. However, a comparison of Rust against my current choice of Racket shows clear opportunities regarding bug-catching, text editor choice, career impact, and community. As educators, we must follow through with a software infrastructure that allows this potential to be realized for our students.

Acknowledgments

Thanks to Matthew Ahrens at WPI for extensive feedback on a draft.

References

- [1] ACM and IEEE. Computing curricula 2020: Paradigms for global computing education, 2021.
- [2] Albert Bandura. Personal and collective efficacy in human adaptation and change. *Advances in psychological science*, 1, 1998.
- [3] Hugh Burkhardt and Alan H Schoenfeld. Improving educational research: Toward a more useful, more influential, and better-funded enterprise. *Educational Researcher*, 32(9):3–14, 2003.
- [4] Chen Chen, Paulina Haduong, Karen Brennan, Gerhard Sonnert, and Philip Sadler. The effects of first programming language on college students' computing attitude and achievement: a comparison of graphical and textual languages. *Computer Science Education*, 29(1):23–48, 2019.
- [5] The RLS Developers. Rust support for Visual Studio Code. <https://github.com/rust-lang/vscode-rust>. Accessed Aug. 15, 2022.
- [6] Onyeka Ezenwoye. What language?-the choice of an introductory programming language. In *2018 IEEE Frontiers in Education Conference (FIE)*, pages 1–8. IEEE, 2018.
- [7] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. *How to design programs: an introduction to programming and computing*. MIT Press, 2018.
- [8] Robert Bruce Findler. DrRacket: The Racket programming environment. *Racket Language Documentation*, 2014.
- [9] Robert Bruce Findler, John Clements, Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, Paul Steckler, and Matthias Felleisen. DrScheme: A programming environment for Scheme. *Journal of Functional Programming*, 12(2):159–182, 2002.
- [10] Center for Evaluating the Research Pipeline. Data buddies survey 2020 department report. *Computing Research Association*, 2021.
- [11] Tony Hoare. Null references: The billion dollar mistake. QCon London, 2009.
- [12] Karyn L Lewis, Jane G Stout, Noah D Finkelstein, Steven J Pollock, Akira Miyake, Geoff L Cohen, and Tiffany A Ito. Fitting in to move forward: Belonging, gender, and persistence in the physical sciences, technology, engineering, and mathematics (pstem). *Psychology of Women Quarterly*, 41(4):420–436, 2017.
- [13] David Liben-Nowell and Anna N. Rafferty. Student motivations and goals for CS1: themes and variations. In Larry Merkle, Maureen Doyle, Judithe Sheard, Leen-Kiat Soh, and Brian Dorn, editors, *SIGCSE*, pages 237–243. ACM, 2022.
- [14] Andrew Luxton-Reilly, Ibrahim Albluwi, Brett A Becker, Michail Giannakos, Amruth N Kumar, Linda Ott, James Paterson, Michael James Scott, Judy Sheard, and Claudia Szabo. Introductory programming: a systematic literature review. In *Proceedings Companion of the 23rd Annual ACM Conference on Innovation and Technology in Computer Science Education*, pages 55–106, 2018.
- [15] Shing Lyu. *Welcome to the World of Rust*, pages 1–8. Apress, Berkeley, CA, 2020.
- [16] Guillaume Marceau, Kathi Fisler, and Shriram Krishnamurthi. Measuring the effectiveness of error messages designed for novice programmers. In *SIGCSE*, pages 499–504, 2011.
- [17] Guillaume Marceau, Kathi Fisler, and Shriram Krishnamurthi. Mind your language: on novices' interactions with error messages. In *SPLASH Onward!*, pages 3–18, 2011.
- [18] Junya Nose, Youyou Cong, and Hidehiko Masuhara. A DSL for providing feedback on htdp-based programming. In *TFPIE*, 2021. Submitted. Accessed via ResearchGate.

- [19] Kris Powers, Paul Gross, Steve Cooper, Myles F. McNally, Kenneth J. Goldman, Viera K. Proulx, and Martin C. Carlisle. Tools for teaching introductory programming: what works? In Doug Baldwin, Paul T. Tymann, Susan M. Haller, and Ingrid Russell, editors, *SIGCSE*, pages 560–561. ACM, 2006.
- [20] James Prather, Raymond Pettit, Kayla Holcomb McMurry, Alani L. Peters, John Homer, and Maxine S. Cohen. Metacognitive difficulties faced by novice programmers in automated assessment tools. In Lauri Malmi, Ari Korhonen, Robert McCartney, and Andrew Petersen, editors, *ICER*, pages 41–50. ACM, 2018.
- [21] Norman Ramsey. On teaching *How to Design Programs*: observations from a newcomer. In Johan Jeuring and Manuel M. T. Chakravarty, editors, *ICFP*, pages 153–166. ACM, 2014.
- [22] Rosemary S Russ. Epistemology of science vs. epistemology for science. *Science Education*, 2014.
- [23] Ferrous Systems. rust-analyzer. <https://rust-analyzer.github.io/>. Accessed Aug. 15, 2022.
- [24] Darryl K. Taft. Microsoft VS code: Winning developer mindshare. <https://www.techtarget.com/searchsoftwarequality/news/252496429/Microsoft-VS-Code-Winning-developer-mindshare>, 2021. Accessed Aug. 14, 2022.
- [25] Yoshiki Takashima, Ruben Martins, Limin Jia, and Corina S. Pasareanu. SyRust: automatic testing of rust libraries with semantic-aware program synthesis. In Stephen N. Freund and Eran Yahav, editors, *PLDI*, pages 899–913. ACM, 2021.
- [26] Rust Team. Code of conduct. <https://www.rust-lang.org/policies/code-of-conduct>, 2022. Accessed Aug. 14, 2022.
- [27] TIOBE. TIOBE index. <https://www.tiobe.com/tiobe-index/>, 2022. Accessed Aug. 14, 2022.
- [28] Aaron Weiss, Daniel Patterson, Nicholas D. Matsakis, and Amal Ahmed. Oxide: The essence of Rust. *CoRR*, abs/1903.00982, 2019.
- [29] Chris Wilcox. The role of automation in undergraduate computer science education. In *SIGCSE*, pages 90–95, 2015.

An Online Debugging Tool for Rust-based Operating Systems

Zhiyang Chen, Ye Yu, Zhengfan Li, Jingbang Wu
School of Computer Science and Engineering
Beijing Technology and Business University
<wujingbang@btbu.edu.cn>

Rust-Edu Workshop 2022

<https://rust-edu.org/workshop>

Abstract

Convenient source code level debugging tools are vital for monitoring and understanding complicated kernel code, especially system call interactions between user and kernel mode. Robust Rust language debugging tools help experiment and develop Rust-based Operating Systems. However, existing RISC-V and Rust experimental environments are challenging to build and get started, which is not conducive to beginners' kernel learning and development efforts. This project proposes to implement a remote debugging tool for rust kernel based on Visual Studio Code and cloud servers: deploying QEMU virtual machines emulating rust-based Operating System in the cloud server, connecting with user's web browser through gdbserver provided by QEMU, implementing remote single-step breakpoint debugging capability, and providing a user-friendly debugging approach for rust kernel and user mode programs' source code.

1 Introduction

rCore-Tutorial-v3[1] is a Unix-like educational Operating System running on RISC-V platforms written in Rust. We found that most students have faced two problems when learning about this Operating System:

The first problem is the clunky, time-consuming environment configuration for QEMU, rust toolchains, and their dependencies. The second is an inconvenience in debugging: QEMU supports gdb debugging, but its text-based user interface is unfriendly to beginners and inconvenient for going through the code.

To address these two problems, we have implemented an online debugging system¹. By using this online debugging system, an operating system can be written and debugged through a web browser. There are already similar products, e.g., Github codespaces. However, most of them are not open-sourced, so it's restrictive to use and deploy them, and also, they do not have OS-related debugging capabilities.

The advantages of online debugging are that this approach does not require high-performance local computers, does not require local configuration of the development environment, is easy for collaboration, and can build an efficient platform for teaching Operating System development.

In recent years, highly customizable lightweight IDEs, such as Sublime Text, Atom, and Visual Studio Code, have gained popularity rapidly.[6] However, lightweight IDEs have minimal support for online operating system debugging. Given the growing popularity of lightweight IDEs and our observation of the lack of online operating system support, we designed and implemented an online operating system debugging environment based on Visual Studio Code, one of the most popular lightweight IDEs.

2 Design and Implementation

2.1 Overall Architecture Design

The online debugging system we designed implements remote debugging capability for operating systems running on QEMU or real hardware by separating the debugger and the debuggee kernel. The debuggee kernel runs on the server, and the user sends debugging-related requests to it, as shown in the Figure 1.

¹Github: <https://github.com/chenzhiy2001/code-debug>

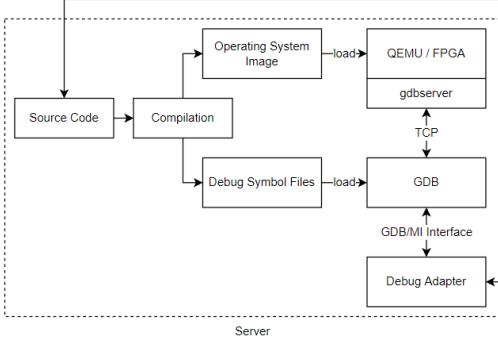


Figure 1: Overall Architecture Design.

Figure 2: Debug Adapter.

In Figure 1, the Source Code is the source code of the OS to be compiled. When the user sends a compilation request, the rust toolchain in the server will compile the OS' source code with a specific compilation configuration and generate an image file and a debug information file that meets the OS' debugging requirements. If the user sends a debug request, QEMU running in the server or FPGA connected to the server will load the image file (We use QEMU as an example in this paper). GDB in the server will load the debug information file and connect to the gdbserver of QEMU.

Extension Frontend in the figure is the module running in Web-Based IDE, which is responsible for debugging related functions. Debug Adapter[2] is a separate process running in the server which is responsible for handling the requests sent by Extension Frontend. When GDB successfully loads the debug information file and connects to Qemu's gdbserver, the Debug Adapter process starts and waits for requests from Extension Frontend. The Debug Adapter will convert the request to GDB commands and send it to GDB. GDB returns the information in GDB/MI Interface[7] to the Debug Adapter after executing the GDB command. Debug Adapter returns the result to Extension Frontend after parsing.

When Extension Frontend receives messages from Debug Adapter, it converts them into Debug UI update messages and sends them to Debug UI and Text Editor module on Web-Based IDE. Likewise, Debug UI and Text Editor can send messages to Extension Frontend, such as breakpoint update messages.

To complete the above process, openvscode-server[3], debug plugin, QEMU, GDB, and rust toolchain must be installed on the server. Users can install them manually or use our docker container containing the above tools to get rid of the trouble of configuration.

Next, we introduce this remote debugging tool in two parts, server side and Web-Based IDE.

2.2 Server Side

OpenVSCode Server is a fork of VSCode that adds a server layer to VSCode's five-layer architecture, allowing it to run in a browser and communicate with a web server that provides a remote development environment. Currently, we use OpenVSCode Server directly as the Web Interface.

After a user accesses the remote debugging tool's website through a web browser, the server returns a web version of Visual Studio Code. The web version of Visual Studio Code is implemented by openvscode-server open-source software.

Users can edit the source code stored in the server and use the remote terminal in the web version of Visual Studio Code. We have already configured the QEMU, GDB, and rust toolchain in the server . Users can debug manually by using QEMU, GDB, and other tools through the remote terminal, or the debugging plugin on the web version of Visual Studio Code.

Users can follow the documentation in our Github repository to install OpenVSCode Server, rust toolchain, QEMU, rCore-Tutorial, and GDB manually or using dockerscript provided in the repository.

Suppose the user chooses to debug with the debug plugin. In that case, the first step the debug plugin does is to compile the kernel and get the executable image file and the debug information file for debugging. The following is an example of the rCore-Tutorial-v3 operating system to describe how to get the debug information files of the kernel.

With the default compilation configuration, the OS image and debug information files generated by compiling rCore-Tutorial-v3 are unuseable for OS debugging because the compiler makes many optimizations and removes most of the symbolic information for debugging. Therefore, we need to modify the compilation configuration files so that the compiler does not optimize the code and does not remove debugging information as much as possible.

In general, rust projects created by cargo can be compiled in both release and debug modes, where release mode optimizes the code at a higher level and removes debug-related information. In contrast, debug mode optimizes the code at a lower level and keeps debug-related information, which is more suitable for our needs. However, rCore-Tutorial-v3 does not support building in debug mode. So, to debug successfully, it is needed to modify the configuration file for release mode to turn off code optimization and keep debugging information.

In addition, rCore-Tutorial-v3 discarded the ".debug_info" and other segments in the user program linker script linker.ld to improve performance, and modifying the linker script makes the linker not to ignore these debug information segments. The linker script can be modified so that the linker does not ignore these debugging information fields. Next, clear the compiled files to make the changes to the link script takes effect.

After keeping the debug information required for debugging, the disk space occupied by the user program becomes larger, which leads to crash and overflow of the easy-fs file system used in rCore-Tutorial-v3, so it is also necessary to change the relevant configuration of the easy-fs-fuse disk packing program for a bigger virtual disk size. In addition, the memory occupied by the user program when running will also increase, so the user stack and kernel stack size also need to be adjusted.

This project turns these changes on configuration files, link scripts, and operating system source code into a diff file. The user only needs to apply this diff file in a remote terminal with a git command to complete the above changes. In the debugging tool's documentation, we provide a link to an already patched rCore-Tutorial so that students don't have to patch rCore-Tutorial themselves.

After compilation, QEMU on the server loads the OS image and opens a gdbserver. Then, GDB loads the symbolic information file and connects to the gdbserver provided by QEMU.

Debug Adapter is a process responsible for coordinating the code editor with the debugger (which is GDB in this project). When GDB is ready, the Debug Adapter process starts and waits for debugging requests sent from the Extension Frontend module in the user's Web-Based IDE.

Once the Debug Adapter receives a request, it converts the request (Debug Adapter Requests) into text that conforms to the GDB/MI interface specification and sends it to GDB. In addition, events such as privilege level changes and breakpoint activation during debugging are also returned to the Extension Frontend through Event messages.

2.3 Web-Based IDE

Extension Frontend runs on the user's Web-Based IDE and communicates with the Debug Adapter on the server. It listens to the messages received and sent by the Debug Adapter and sends Requests, corresponding Responses, and Events. Extension Frontend parses the received Requests and Events and forwards the required information to the Debug UI. If the Debug UI passes a message to the Extension Frontend, the Extension Frontend will also convert the message into Requests and send it to the Debug Adapter.

In addition to the existing native UI of VSCode, the Web-Based IDE invokes a new tab to provide a richer user interface for displaying register values, memory data, privilege mode and breakpoint group through the WebView capability provided by Visual Studio Code.

Using front-end technologies such as bootstrap, jQuery, and CSS, we split debugging information into three sub-tabs and placed five function buttons in a floating widget for a more friendly, intuitive and efficient user

interface.

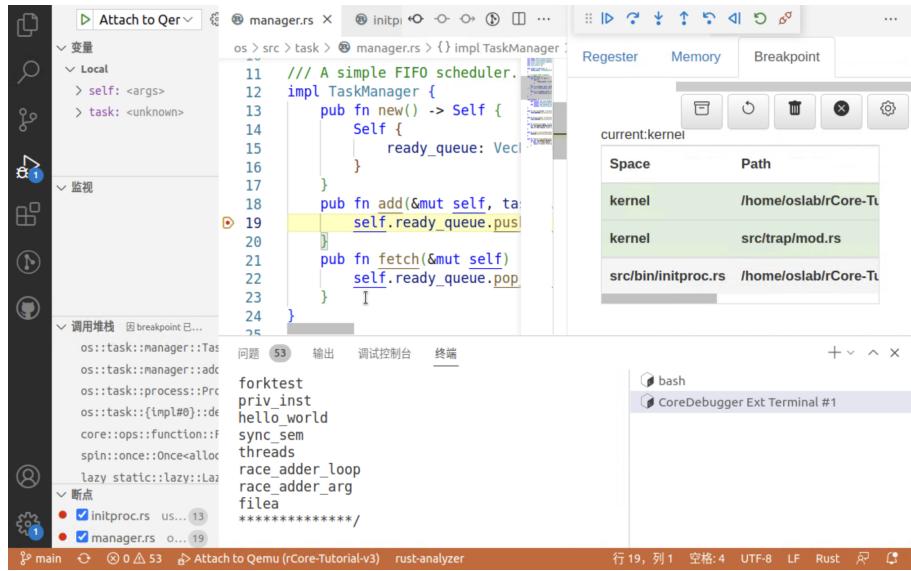


Figure 3: A Screenshot of the User Interface.

2.4 Breakpoint Conflicts in Kernel and User mode

The key problem in implementing the debugging functionality of the operating system is setting the breakpoints of the kernel mode and the user mode simultaneously. However, the breakpoint settings of the user mode and kernel modes are conflicting. The reasons are that GDB sets breakpoints based on memory addresses, but the TLB is refreshed when the kernel mode switches to user mode. For example, When the rCore-Tutorial-v3 operating system is running in kernel mode, if GDB is used to set a user-mode program breakpoint, this user-mode breakpoint will not be hit. The reason is that the sfence.vma instruction of the risc-v processor is executed when the privilege level is switched so the TLB is refreshed into the page table of the user process, causing the breakpoint previously set in the kernel address space to fail.

The main idea in solving this problem is to cache those breakpoints that cause breakpoint conflict, and when the time is right, let GDB set these cached breakpoints. When running in user mode, cache kernel mode breakpoints; when running in kernel mode, cache user mode breakpoints. We have added a breakpoint group management module to the Debug adapter to implement this functionality.

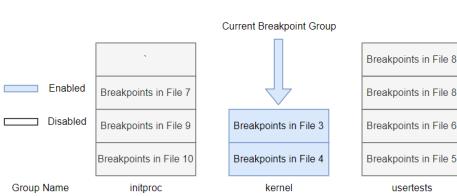


Figure 4: Breakpoint Group.

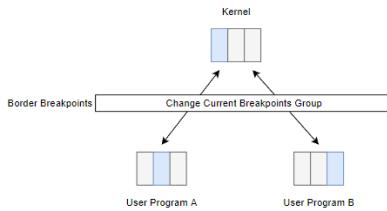


Figure 5: Breakpoint Group Switching.

The breakpoint group management module uses a dictionary to cache all breakpoints set by the user (including kernel mode and user mode breakpoints). The key of an element in the dictionary is the code name of the memory address space, and the element's value is the breakpoint group corresponding to the code name, that is, all breakpoints in this memory address space. When any breakpoint is hit, the Debug Adapter will try to

detect which breakpoint group the currently hit breakpoint belongs to. The breakpoint group that contains the latest hit breakpoint is called the Current Breakpoint Group.

When the user sets a new breakpoint in the Web-Based IDE, the Debug Adapter will receive a Request to set a breakpoint. The breakpoint group management module in the Debug Adapter will first store the breakpoint information in the corresponding breakpoint group and then evaluate whether the breakpoint group that this breakpoint belongs to is the current breakpoint group and, if so, let GDB set this breakpoint immediately. If not, this breakpoint should not be set by GDB right now.

Under such a caching mechanism, GDB will not set breakpoints in kernel mode and user mode at the same time, thus avoiding the breakpoint conflict between kernel mode and user mode. Next, a mechanism is needed to switch the breakpoint group at the right time to ensure that GDB sets a specific breakpoint before it may be hit. Exploiting the timing of privilege level switching is ideal. This project sets breakpoints when the kernel mode enters the user mode and the user mode returns to the kernel mode. We call these two breakpoints border breakpoints. If these two breakpoints are hit, the privilege level has switched, and then the memory address space has also switched, so the breakpoint group should also be switched. We make the Debug Adapter check whether the breakpoint is a border breakpoint every time a breakpoint is hit. If so, remove all breakpoints in the old breakpoint group before setting breakpoints in the new breakpoint group. As Figure 5 shown.

To ensure that the above functionalities work as expected, when the breakpoint group is switched, the symbol table file should also be switched accordingly.

2.5 Example: Current Privilege Mode Detection

One of the significant differences between debugging operating systems and general programs is that when debugging an operating system, the user often needs to pay attention to what privilege mode the CPU is currently at. Therefore, operating system debugging tools need the ability to detect the current privilege level. We take privilege mode detection as an example to show this operating system's online debugging system's typical process.

The Debug Adapter then responds to these requests by sending commands to GDB. RISC-V processors have no registers to reveal the current privilege level, so the current privilege level cannot be obtained directly through the "info registers" GDB command. The Debug Adapter will try to obtain the memory address and file name of the most recently hit breakpoint and then determine the current privilege level.

After getting the current privilege level, the Debug Adapter returns Responses to the Extension Frontend. Extension Frontend receives and parses Responses and Events, passing the information to the Debug UI. The Debug UI updates the interface after receiving the information.

3 Summary and Outlook

The main work of this project is to extend the source code-level tracking and analysis capabilities of the Rust language and operating system kernel features based on the existing debugger plugin of the Visual Studio Code editor. We have implemented: data acquisition of key registers and memory values, support for setting user-mode programs' breakpoints when running in kernel, accurate acquisition of current privilege mode, and automatically replacing symbol table files.

Currently, Some students, including participants of an OS contest [5] found the tool helpful in debugging, so they tried to use it while writing their own OS. Other students found the tool has drawbacks and tried to improve it to make it more convenient.

Due to the limitations of GDB's support for the rust language, some language elements, such as Self variable, Vec, VecDeque, and lazy_static macro are not supported[4], which causes some kernel data like PCB untrackable. We hope to fix those problems next.

We hope to continue developing this set of online debugging tools to support inspecting these language elements so that more kernel data structures can be presented.

References

- [1] rCore-Tutorial-v3. Available online: <https://github.com/rcore-os/rcore-tutorial-v3>.
- [2] Specification of debug adapter protocol. Available online: <https://microsoft.github.io/debug-adapter-protocol/specification>.
- [3] Openvscode-server. Available online: <https://github.com/gitpod-io/openvscode-server>.
- [4] Documentation of GDB. Available online: <https://sourceware.org/gdb/onlinedocs/gdb/rust.html>.
- [5] CSCC OS Contest. Available online: <https://os.educg.net/2022cscc>.
- [6] Hongfei Fan, Kun Li, Xiangzhen Li, Tianyou Song, Wenzhe Zhang, Yang Shi, and Bowen Du. Covscode: A novel real-time collaborative programming environment for lightweight ide. *Applied Sciences*, 9(21):2, October 2019.
- [7] Stan Shebs Richard M. Stallman, Roland Pesch. *Debugging with GDB: The GNU Source-Level Debugger*. Free Software Foundation, 9 edition, 2002.

The Pluggable Interrupt OS: Writing a Kernel in Rust

Gabriel J. Ferrer

Department of Mathematics and Computer Science

Hendrix College

<ferrer@hendrix.edu>

Rust-Edu Workshop 2022

<https://rust-edu.org/workshop>

Abstract

As Rust is a suitable language for writing bare-metal code, it is a viable competitor with C and C++ as a programming language of instruction for the Operating Systems course. The popular blog Writing an OS in Rust [5] provides stellar resources for instructors wishing to teach students how to create a kernel using Rust. In this paper, I describe a crate I developed called the Pluggable Interrupt OS [4]. Building on the resources from the aforementioned blog Writing an OS in Rust, this crate enables students to easily create a functioning kernel for the x86 architecture. A key aspect of its simplicity is presenting students with an API that renders the `unsafe` keyword unnecessary by presenting straightforward abstractions for setting up interrupts. Several assignments employed in the classroom built upon this crate are described.

1 Introduction

The Rust programming language is an appealing language of instruction for an Operating Systems course. Most Operating Systems courses use the C or C++ languages. Like these languages, a Rust executable can run on the bare metal in the absence of an operating system, thus enabling its use to create one. David Evans gives a persuasive argument for the superiority of Rust over other alternatives, particularly C and C++, for an Operating Systems course. [2].

The blog Writing an OS in Rust [5] gives lots of specific details for writing Rust code that interacts with x86 hardware, specifically the VGA buffer and the interrupt handlers. In terms of abstraction, the blog presents code to implement the `print!` and `println!` macros using the VGA buffer, but it presents no additional abstractions for setting up interrupt handlers.

In order to facilitate student projects by providing useful abstractions for bare-metal programming, I created the Pluggable Interrupt OS crate [4]. My goals for the crate are as follows:

- Enable students to create a 100% Rust bare-metal program.
- No `unsafe` blocks in student code.
- Provide straightforward abstractions for setting up timer and keyboard interrupts, as well as kernel startup code and background code.

The rest of this paper is organized as follows. We begin by describing the curricular context in which this crate was developed. Next, we examine the `HandlerTable` struct, the abstraction provided by the crate for setting up handlers for interrupts, startup, and background code. We next examine the setup of a cooperative multitasking kernel that shows how to integrate interrupt code with background code. We then discuss our classroom experiences and conclude with a discussion of future directions for the crate.

2 Curricular Context

Our curriculum requires all Computer Science majors and minors to take a two-course introductory sequence: Foundations of Computer Science and Data Structures. Once students complete Data Structures, they can

take any of our upper level courses, including Operating Systems. Students learn Python in Foundations of Computer Science and Java in Data Structures. Operating Systems is the only course in which we teach Rust.

Prior to attempting bare-metal programming, students gain experience with Rust over a series of assignments of gradually increasing difficulty and sophistication. In the first set of assignments, students implement each of the following Unix command line utilities as a Rust executable:

- `ls` - list a directory
- `rm` - remove a file
- `mv` - rename a file
- `cp` - copy a file
- `head` - see the first n lines of a file
- `wc` - count words, lines, and characters in a file
- `fgrep` - search for a text string in a file
- `sort` - sort the lines of a file
- `wget` - download a web page

Students then build two larger programs in Rust (inspired by assignments from David Evans's course [3]):

- Unix shell, including pipelines, I/O redirection, and background processes.
- Web server, spawning a separate thread for each HTTP request.

3 Setting Up Interrupt Handlers

As the primary goal of the crate is to enable students to easily set up interrupt handlers, the `HandlerTable` data type, shown in Figure 1, is provided to make it possible to do so clearly and concisely. Its four attributes control all programmer-specified aspects of the system's behavior:

- The `timer` function, if present, is invoked by the timer interrupt handler.
- The `keyboard` function, if present, is invoked by the keyboard interrupt handler.
- The `startup` function, if present, is invoked when the system begins.
- The `cpu_loop` function runs continuously whenever there is no interrupt to be handled. (If no function is specified, it executes the x86 `hlt` instruction.)

```
pub struct HandlerTable {  
    timer: Option<fn()>,  
    keyboard: Option<fn(DecodedKey)>,  
    startup: Option<fn()>,  
    cpu_loop: fn() -> !  
}
```

Figure 1: The `HandlerTable` data type

None of the projects listed in Section 2 requires students to use the `unsafe` keyword. Teaching students to make disciplined use of `unsafe` in addition to the other complications involved with bare-metal programming

was more material than I felt possible to cover in the time available. This is why building the crate to abstract away from `unsafe` was an important goal.

As the crate's interrupt handling code is under 200 lines of code, it is feasible to explain it to students in detail. Here are three examples of using `unsafe` to interact with hardware that are straightforward to explain during class:

- Invoking the instruction to assign the interrupt handlers to the interrupt controller must take place within an `unsafe` block.
- Reading the key that triggered a keyboard interrupt requires accessing a specific memory location. That memory access can only occur within an `unsafe` block.
- Students can see where the `timer` and `keyboard` functions from `HandlerTable` are invoked, followed by the `unsafe` code to inform the interrupt controller that the handler function has completed its work.

Figure 2 shows a “Hello, World!” employment of the crate (included as `main.rs`). It prints a `.` character on each `timer` interrupt, and prints each character typed on a `keyboard` interrupt. Using the builder pattern, the programmer specifies the handler function for each attribute, starting execution with the `.start()` method.

```
fn startup() {println!("Hello, world!");}

fn tick() {print!(".");}

fn key(key: DecodedKey) {
    match key {
        DecodedKey::Unicode(character) => print!("{}", character),
        DecodedKey::RawKey(key) => print!(":{}?", key),
    }
}

#[no_mangle]
pub extern "C" fn _start() -> ! {
    HandlerTable::new()
        .keyboard(key)
        .timer(tick)
        .startup(startup)
        .start()
}
```

Figure 2: Hello, World!

The `.start()` method does the following:

- Runs the `startup` function.
- Stores the `HandlerTable` in a spin lock `Mutex`, accessible to the low-level interrupt handling code.
- Initializes and starts the timer and keyboard interrupts.
- Runs the `cpu_loop` function.

Once `.start()` has completed, the `cpu_loop` code runs indefinitely, unless an interrupt handler is triggered. Once a handler is triggered, control transfers to the handler function. Once the handler function exits, control resumes where it left off in the `cpu_loop` code. Note that in the Figure 2 example, since there is no `cpu_loop` code, all activity happens solely in the interrupt handlers.

4 Integrating Timer, Keyboard, and Ongoing Computation

Because interrupts are invoked asynchronously, it is necessary to have a strategy for concurrent access to data structures shared by the handlers and CPU code. Figure 3 shows one strategy for integrating timer and keyboard events with an ongoing computation. This is part of the skeletal code for one of the course assignments, in which students create a cooperative multitasking kernel with four processes in the VGA buffer, as shown in Figure 4. `AtomicCell` objects (from the `crossbeam` crate [1]) wrap both the most recent keystroke as well as the count of timer interrupts. The `cpu_loop()` function has a `Kernel` object as a local variable that can be given a keystroke, asked to draw itself, or asked to execute an instruction. It constantly checks the `AtomicCell` objects that store the ticks and recent keystrokes, and updates the `Kernel` accordingly.

```
static ref LAST_KEY: AtomicCell<Option<DecodedKey>> = AtomicCell::new(None);
static ref TICKS: AtomicCell<usize> = AtomicCell::new(0);

fn cpu_loop() -> ! {
    let mut kernel = Kernel::new();
    let mut last_tick = 0;
    kernel.draw();
    loop {
        if let Some(key) = LAST_KEY.load() {
            LAST_KEY.store(None);
            kernel.key(key);
        }
        let current_tick = TICKS.load();
        if current_tick > last_tick {
            last_tick = current_tick;
            kernel.draw();
        }
        kernel.run_one_instruction();
    }
}

fn tick() {TICKS.fetch_add(1);}

fn key(key: DecodedKey) {LAST_KEY.store(Some(key));}
```

Figure 3: Integrating Timer and Keyboard with Background Computation

5 Classroom Assignments

Students were given three assignments in which to use the Pluggable Interrupt OS. The first assignment was to create a simple bare-metal video game. The second assignment was to create a video game kernel in which the user picks a student-created game from the previous assignment to play. Students learn about process management as the user can pause games, start new instances of games, and resume games.

The third assignment was to create the four-window cooperative multitasking operating system depicted in Figure 4, building on the code template introduced in Section 4. Two applications are provided: a program to compute π via a power series, and an interactive program to compute an average. The first application is designed for running in the background, while the second emphasizes I/O. The student is to divide the VGA buffer into four windows, each of which can run one of the two applications.

About half of the students in the course successfully completed the video game and video game kernel assignments, and about a third of the students completed the cooperative multitasking operating system. There seem to have been two major obstacles to completing the assignments:

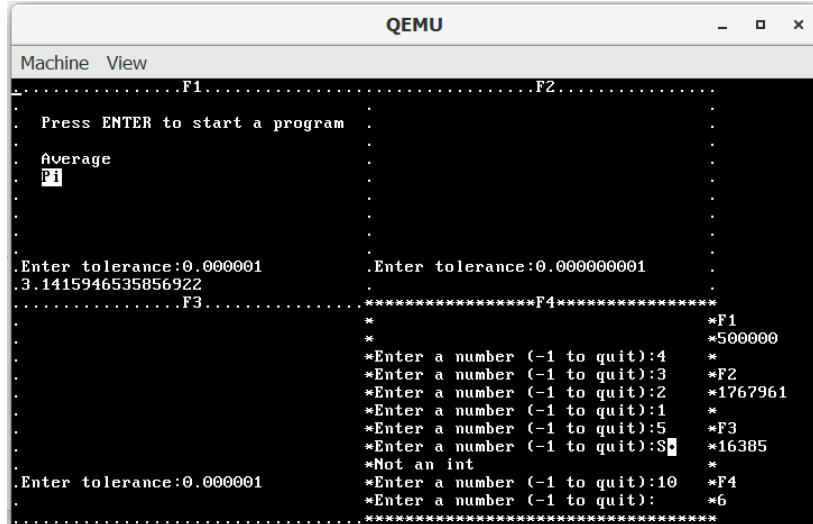


Figure 4: Screenshot of Cooperative Multitasking Kernel

- It was possible to earn a good grade in the course without successfully completing them.
- In spite of the simplifications provided by the Pluggable Interrupt OS, the code template for the cooperative multitasking assignment was still over 200 lines of code. The code template over-specified the design; next time, I plan to provide less code and create more conceptual space for students to work out their own designs.

6 Conclusion

The overall design goals for the Pluggable Interrupt OS crate were met. However, successful classroom implementation may require further effort to improve explanation of the assignments. Changing the course grading criteria to more strongly incentive the effort necessary to write the kernel projects may also be helpful.

The crate's capabilities do enable students to create a bare-metal program that can legitimately be considered a kernel. But two key additional capabilities would expand pedagogical options considerably:

- Mechanisms to enable pre-emptive multitasking.
- Introduce a mechanism for persistent storage, thus enabling the creation of a file system.

References

- [1] Alex Crichton, Jeehoon Kang, Aaron Turon, and Taiki Endo. `crossbeam`. <https://crates.io/crates/crossbeam>. Retrieved August 11, 2022.
- [2] David Evans. Using Rust for an undergraduate OS course. <http://rust-class.org/0/pages/using-rust-for-an-undergraduate-os-course.html>, 2013. Retrieved August 20, 2022.
- [3] David Evans. cs4414: Operating systems. <http://rust-class.org/>, 2014. Retrieved August 20, 2022.
- [4] Gabriel Ferrer. Pluggable interrupt OS. https://crates.io/crates/pluggable_interrupt_os. Retrieved August 20, 2022.
- [5] Philipp Oppermann. Writing an OS in Rust. <https://os.phil-opp.com>. Retrieved August 20, 2022.

Experience Report: Two Semesters Teaching Rust

Matthew Fluet

Department of Computer Science

Rochester Institute of Technology

<mtf@cs.rit.edu>

Rust-Edu Workshop 2022

<https://rust-edu.org/workshop>

Abstract

In each of the spring semesters of AY2020/21 and AY2021/22, I have taught a semester long course on Safe and Secure Systems Programming in Rust. Overall, the experience has been quite positive. In this experience report, I describe the structure of the course, the lectures that have worked well and those that remain a struggle, some of the programming assignments, and a qualitative comparison of the course to a similar course that I have taught on Functional Programming and Haskell.

1 Introduction

The Computer Science program at the Rochester Institute of Technology has for many years taught a series of courses as instances of the umbrella co-listed BS/MS courses CSCI-541: Programming Skills and CSCI-641: Advanced Programming Skills, with the goal “to introduce the students to a programming paradigm and an appropriate programming language chosen from those that are currently important or that show high promise of becoming important”. Regularly offered instances of the course have included Functional Programming and Haskell, Efficient Design in Modern C++, and Design Patterns and C#/.NET.

Recognizing that Rust clearly falls into the category of programming languages “that are currently important or that show high promise of becoming important”, in February 2020, I proposed a new instance of the course titled Safe and Secure Systems Programming in Rust. At the time, I knew of the language through technical news outlets, social media, and the rare academic paper, but had no first-hand experience programming in Rust. The subsequent COVID-19 pandemic limited the amount of time that I had hoped to devote to advanced preparation. Nonetheless, I was able to offer the course instance in both Spring AY2020/21 [14] and Spring AY2021/22 [15] with an in-person instruction mode.

In this experience report, I describe the structure of the course, some of the lecture topics that have worked well and those that remain a struggle, and a qualitative comparison to the Functional Programming and Haskell instance of Programming Skills that I have also regularly taught. An appendix describes the major programming assignments used in the course.

2 Course Structure

As noted above, the CSCI-541: Programming Skills and CSCI-641: Advanced Programming Skills courses are co-listed, the former an upper-level course in the BS program and the latter a course in the MS program. Teaching Rust as an (Advanced) Programming Skills instance is somewhat of a “best-case scenario”, in the sense that the course prerequisites ensure that undergraduate students have completed the entire introductory programming sequence (1 semester Python, 1 semester Java, 1 semester C) as well as CSCI-344: Programming Language Concepts (and are typically final year students) and that graduate students have completed the bridge courses (1 semester Python, 1 semester Java). It is safe to assume that the students have had a non-trivial amount of programming experience under multiple languages.

The general structure of the course is fairly standard: lectures with discussion and live coding, six programming assignments, and independent team projects. In addition, there is an “Assignment Design and Critique” activity, where students (solo or pairs) are asked to design a novel programming assignment for the first half of the

course (Rust Basics) and to review and critique the assignments of others, and a “Research and Blog Post” activity, where students (solo or pairs), research and blog about an interesting advanced aspect of the language suitable for the second half of the course (Advanced Concepts) and review and critique the blog of others.

The first half of the course focuses on learning the basics of Rust by reading “the book” (*The Rust Programming Language* [22, 23] by Steve Klabnik and Carol Nichols, with contributions from the Rust Community). We cover nearly the entire book in order, except that we omit “14. More about Cargo and Crates.io”, skim “18. Patterns and Matching” when considering “6. Enums and Pattern Matching”, consider “17. Object Oriented Programming Feature of Rust” (and the “Advanced Traits”, “Advanced Types”, and “Advanced Functions and Closures” sections of “19. Advanced Features”) before considering “16. Fearless Concurrency” and “20. Final Project: Building a Multithreaded Web Server” together.

The second half of the course focuses on different advanced topics by watching videos, reading blogs, and reading research papers and independent team projects. We consider Rayon [26, 27, 35] (simple work-stealing parallelism for Rust), Macros [37, 5], `async/await` [21, 3] and Futures [43, 42, 3], Unsafe Rust [37, 38], Rust and the Web (Servo Web Browser Engine [1] and Stylo/Quantum CSS Engine [18, 30]), Session Types [24]/Typestates [9], Foreign Function Interfaces and NoStd, Rust for Operating Systems [11, 25], and Polonius [28] (alias-based formulation of the borrow checker).

The course is scheduled with 75min lectures twice weekly. Each lecture topic includes some preparation work (readings from *The Rust Programming Language* [22, 23] for the first half; other documentation (*The Rustonomicon* [38], *Asynchronous Programming in Rust* [3], etc.), blog posts, or research papers for the second half), along with a pass (done)/fail (not done) preparation work “quiz” to demonstrate basic understanding of the material made up of technical questions and open-ended concept questions. Although originally conceived for the first offering of the course as a means of maintaining the pace should any (hastily prepared) lectures have serious issues, the quiz questions and responses have served as excellent discussion starters. Some particularly good discussions arose from questions like:

- Review the “Invalid Array Element Access” sub-section [22, Chapter 3.2]. What do you think about the assertion “In many low-level languages, this kind of check is not done”? What implications does this have for Rust’s execution of an array element access operation? Does this change your opinion of Rust as a “low-level language”?
- Rust does not have throwable/catchable exceptions. What are your concerns about programming in a language without exceptions? Do you think that Rust’s error-handling features will help you write more robust code?
- Should the `Rc` type provide a method `fn unwrap(self: Rc<T>) -> T` to recover ownership of the value being reference counted? If not, is there a reasonable alternative?
- Are there any object-oriented design patterns that you have used that would be difficult to translate into Rust? Do you think that there are reasonable alternatives that could be used in Rust?
- The typestate pattern is an API design pattern that encodes an object’s run-time state in its compile-time type. Can you describe another application or domain where the typestate pattern could be employed?
- A cynical kernel developer claims that Rust doesn’t provide any real benefit in this example [11]. The authors simply refactored the “tricky” parts into another function/method. The apparent elegance of the Rust code can be matched in C by performing the same essential refactoring. What is your response to the cynical kernel developer? Does the Rust version of `consume` provide real protection against some bugs that could be introduced in the C version of `semaphore_consume`? Does the Rust version of `consume` have any inefficiencies that are not shared by the (original or refactored) C version of `semaphore_consume`?

3 Lectures: The Good, the Borrowed, and the Not-so-Pretty

Most lectures are fairly typical collegiate fare: a mix of discussion, question and answer, white-board diagramming, and (more specific to a programming-language course) code review, live coding, and browsing of API documentation.

3.1 The Good

There are three lectures that I think have worked particularly well. Interestingly, two of the three employ the strategy of developing a solution through exploring a number of incorrect attempts.

The first is a lecture on implementing iterators¹. This is also one of the more significant changes that was made to the course between the first and second offering. In the first offering, I focused primarily on the implementation of iterator adaptors and failed to discuss the implementation of a base iterator on a data structure. The vast majority of students failed to implement iterators on a trie in the corresponding programming assignment; in response, I interrupted the course schedule with this lecture and allowed students to resubmit. In the second offering of the course, when this lecture occurred before the corresponding programming assignments, students performed significantly better. The first component of the lecture involves implementing `iter`, `iter_mut`, and `into_iter` on `struct Triple<T>{a: T, b: T, c: T}`. We observe that while using a simple `index: usize` state suffices for implementing `iter`, trying to use the same technique to implement `iter_mut` leads to lifetime errors. After another incorrect attempt, we show how and why a `fields: Triple<&'a mut T>` state and `std::mem::take` can be used to implement the `iter_mut` (and the same technique can be used to implement `into_iter`). We conclude with the observation that, in general, the implementation of `into_iter` is the most restrictive and the implementation technique that works for it can usually be simply modified to work for `iter_mut` and `iter`. The second component of the lecture involves implementing a preorder iterator on `enum BinTree<T> {Leaf, Node(Box<BinTree<T>>, T, Box<BinTree<T>>)}`, where we observe that a recursive preorder traversal would use recursion and an implicit call-stack, which motivates an explicit stack: `Vec<BinTree<T>>` state. This prepares students to implement an iterator on a trie.

The second is a lecture on message-passing concurrency², where we implement a *broadcast channel* abstraction. `bcast::channel` creates a new broadcast channel, returning sender/receiver halves (like `std::sync::mpsc::channel`). All data sent on the Sender will become available on each currently registered Receiver in the same order as it was sent, and no send will block the calling thread. `recv` will block until a message is available. The Sender can be cloned to send to the same broadcast channel multiple times. The Receiver can also be cloned, which registers a new Receiver with the broadcast channel. A cloned Receiver will only receive messages sent after it has been cloned. In developing the abstraction, we observe that we need a shared collection of `mpsc::Senders`, which motivates implementing `bcast::Sender<T>` with a `Arc<Mutex<Vec<mpsc::Sender<T>>>`. We also observe the need for a `T: Clone` trait bound on the `bcast::Sender<T>::send` method and consider how to minimize the number of clones made and how to recognize and cleanup after `bcast::Receivers` that have hung up already. While it is obvious that the `bcast::Receiver` will need a `mpsc::Receiver`, it is less obvious that a `Weak<Mutex<Vec<mpsc::Sender<T>>>` is needed for cloning a `bcast::Receiver`, the Weak to ensure that the `mpsc::Senders` are dropped when all `bcast::Senders` are dropped, even if `bcast::Receivers` remain. Finally, we observe that no special `Drop` implementation is required for `bcast::Sender` or `bcast::Receiver` and explain why there are no `Send` bounds on any of the methods, despite these being methods to send data between threads.

The third is a lecture is on `unsafe`³, where we implement a *gallery* abstraction. A Gallery is a place to which you can donate objects, after which many can look at (i.e., reference) the object, but none can touch (i.e., mutate) it. Objects donated to the gallery are destroyed all at once, when the gallery itself is destroyed; there is no deallocation of individual objects while the gallery itself is still owned. This is essentially a simplified version of the typed-arena crate [44]. A series of attempts, starting with `struct Gallery<T>{objs: Vec<T>}` and ending with `struct Gallery<T>{objs: RefCell<Vec<Pin<Box<T>>>}`, illuminates the need for the various indirections and a small, but necessary, `unsafe { &*res }` to cast a reference with one lifetime to a reference with a larger lifetime. Along the way, we are able to introduce and use Miri [31] to understand (some) of the undefined behavior in the incorrect versions. (Curiously, there are some versions that I feel should be defined behavior, but that Miri nonetheless flags as having undefined behavior.)

¹<https://git.cs.rit.edu/psr2215/notes/-/blob/main/05/iter8ors>

²<https://git.cs.rit.edu/psr2215/notes/-/blob/main/08/bcast>

³<https://git.cs.rit.edu/psr2215/notes/-/tree/main/12/gallery>

3.2 The Borrowed

Rust is well-known for having a vibrant community of Rustaceans popularizing the language. Moreover, this community has produced a significant amount of high-quality content as mdBook online documentation (books, tutorials, etc.), blog posts, and YouTube videos (both conference presentations and self-produced series). All of the course's lecture topics are accompanied by a (sometimes long) list of additional resources.

And, in true Rust fashion, I “borrow” some of that material to supplement lectures. We devote lectures to watching Rayon: Data Parallelism for Fun and Profit [27] by Niko Matsakis (2016 Rust Belt Rust Conference), Procedural Macros vs Sliced Bread [5] by Alex Crichton (2019 RustLatam Conference), The Story of Stylo: Replacing Firefox’s CSS Engine with Rust [30] by Josh Matthews (2017 Rust Belt Rust Conference), and Polonius: Either Borrower or Lender Be, but Responsibly [28] by Niko Matsakis (2019 Rust Belt Rust Conference). Following a semester or more of remote learning, such online content is much less foreign to students. These presentations fit comfortably within the 75min lecture, allowing additional time for reflection and discussion.

3.3 The Not-So-Pretty

Most of the remaining lectures are fair, but not exemplary.

I have not (yet) developed a completely satisfactory introduction to ownership, references, and borrowing. There are aspects of the language that I find work against a clear presentation of the ideas. In most languages with static typing and type inference, we can “dialogue” with the compiler when we do not understand a type error by adding type annotations to confirm or refute our expectations. However, Rust does not have syntax for defining or using a lifetime that is local to a function body, which prohibits annotating examples with the lifetimes inferred by the compiler. Moreover, seemingly in contrast to many other static analyses familiar to students, the borrow checker seems to be characterized more by what it rejects than what it accepts. Thus, examples are littered with documentation of the form “If we uncomment the following line, we get an error.” However, such documentation is not checked, since Rust (like most other languages) does not have a feature to assert that a line of code should have a compile-time error. It is understandable why these aspects of the language exist, given Rust’s commitment to backwards compatibility and the desire to improve the borrow checker to accept more programs. It is also worth emphasizing to students (and to the instructor!) that ownership permeates the language; it is not necessary or expected to have a perfect grasp of the concept after a lecture or two. Rather, it is something that is continually reinforced throughout the course.

My lectures on `async/await` and futures have improved from the first to the second offering of the course. Initially, following much online documentation (such as Asynchronous Programming in Rust [3]), I discussed the Future trait and adaptors before turning to significant examples of `async/await` programming. While this may follow the historical development of the feature, it places the technical details before the motivation. I now start with a high-level description of asynchronous programming in Rust, with the analogy of a (composed) asynchronous task as a widget made up of (nested) gear boxes; we “turn the crank” on the widget until it sticks, at which point we switch to another widget until the “ready light” on the first widget turns on. This suffices for understanding many examples of Async Rust programming. (As an aside, an illustrative in-class demonstration is to use a web server benchmarking tool to compare single-threaded [22, Chapter 20.1], multi-threaded [22, Chapter 20.2], and asynchronous [3, Chapter 9] web servers written in Rust.) Having motivated and used Async Rust, we then turn to the implementation of the Future trait and adaptors⁴ and `async/await` desugaring⁵. The latter has been particularly helpful for demonstrating how `async/await` code compiles to a state machine and why `async/await` must be a language feature (supported by the compiler). Most Async Rust documentation and tutorials assert these facts, without giving a concrete example. I use a simple 13 line `async` function that reads one byte from each of two `&File` arguments and writes the maximum to a `&mut File` argument, which makes use of both parallel (`futures::future::Join`) and sequential (`futures::future::Then`) composition. The final desugaring is over 150 lines, though, and rather easy to get lost in. A simple variation on the function (reading two bytes from a `&File` argument and writing the maximum to a `&mut File` argument) demonstrate how the borrow checker rejects the desugaring. Another variation (opening and creating the files from &`str`

⁴<https://git.cs.rit.edu/psr2215/notes/-/tree/main/11/simple-futures>

⁵<https://git.cs.rit.edu/psr2215/notes/-/blob/main/11/async-desugar>

arguments) hints at the need for pinning, though it remains difficult to discuss the issue of pinning in detail. One observation is that while the community envisions a shiny future [29] for `async` Rust that mostly focuses on how to use `async/await`, rather than on how `async/await` works behind the scenes, the current tutorials and documentation do not yet have the right balance.

4 Comparison with Functional Programming and Haskell

In addition to the Safe and Secure Systems Programming Rust instance of Programming Skills, I have also regularly taught the Functional Programming and Haskell instance, most recently in Fall AY2018/19 [12] and Fall AY2019/20 [13] (and will again in this Fall AY2022/23). Although the courses are similar in spirit, there are some qualitative differences that seem to arise from the differences in the language paradigms and communities. Of course, I should preface this comparison with the acknowledgment that functional programming is my preferred paradigm, that I have studied and programmed in Haskell for significantly longer than I have in Rust, and that I would rate myself as more “fluent” in functional programming idioms than in Rust idioms (although, Standard ML is my “native” functional programming language). When starting the Rust course, I was a little worried that my personal bias would tend towards recreating functional programming idioms in Rust, which I understood to be appropriate to an extent, but I did not want to over do it. I’m pleased to say that that fear was mostly unjustified and I believe that there is almost no overlap in meaningful intellectual content between the two courses.

One of the qualitative differences between Rust and Haskell seems to be the typical size of a “small, interesting function”. This manifests itself in two ways in the courses. First, while my Haskell programming assignments often ask students to implement a dozen or so functions (each with independent tests), my Rust programming assignments often ask students to implement fewer than half that (though some may lend themselves to auxiliary helper functions). One reason for this may be that many of my exercises in Haskell ask students to implement data structures, which is a natural undertaking in Haskell. Conversely, Rust has a rich standard library that one is encouraged to use; moreover, many interesting data structures in Rust (such as those in the standard library) require some use of `unsafe` to be implemented efficiently, but such an approach is not appropriate when teaching Rust. The second way that this manifests in the courses is in the course structure. My Haskell course is scheduled with 50min lectures three times weekly and I devote one lecture at the end of each topic to an in-lecture pair-programming recitation, which works well with “small” functions and I was able to develop without significant effort. It seems much harder to scale down my Rust programming assignments into something that would be suitable for a 50min recitation.

Another qualitative difference between the courses is where to find content for the second half of the course. Both courses are roughly structured with a first half on the language basics (the Haskell course uses *Programming in Haskell* [19] by Graham Hutton) and a second half with more advanced applications of the language. For Haskell, academic conferences (e.g., ICFP, IFL, the Haskell Workshop) and curated collections (e.g., *The Fun of Programming* [16] edited by Jeremy Gibbons and Oege de Moor, *Pearls of Functional Algorithm Design* [4] by Richard Bird) are good sources for (peer-reviewed) papers that present self-contained discussions of a topic with a high-degree of rigor. As noted above, lots of Rust content appears online in the form of blog posts and YouTube videos, but there can be significant variance in the degree of rigor and it is rare to have a supplementary “long form” presentation of the material. On the other hand, blog posts and YouTube videos are much more familiar to BS and MS students than an academic paper (which is one reason that my Haskell course includes a “Research Paper Summary” activity that asks pairs of students to read and summarize an academic paper where Haskell is used as a research vehicle). This may simply be a reflection of the relative ages of the languages, as there is a growing collection of advanced Rust books (e.g., *Rust for Rustaceans* [17] by Jon Gjengset) and more frequent appearances of Rust in academic conferences, though often of the “featherweight Rust” style that tries to give a very formal presentation of some core aspect of the language. But, I believe it does reflect a qualitative difference in the language communities. The “major players” in the Haskell community typically are (or were recently) academics and/or researchers, while they typically are industry practitioners in the Rust community, though this may also be changing in both communities.

A final, and admittedly vague, difference between the courses is the incoming attitudes of the students. Students seem to have a preconceived notion that Haskell is very different from their previous programming experience (although our introductory programming course in Python does emphasize structural recursion). My

impression is that students asking for help with a Haskell assignment more frequently express that they have no idea where to start, while students asking for help with a Rust assignment more frequently have some code that they believe should work and don't understand why it is rejected by the compiler or does not behave as they expect.

5 Conclusion

Overall, my experience teaching Safe and Secure Systems Programming in Rust has been quite positive. Although Rust has a well-deserved reputation for a steep learning curve, students have generally also had a positive experience in the course; from the combined course evaluations (N=26; the total enrollment was 71 students), the “Student learned something of value” question was answered 0% (Strongly) Disagree - 8% Neutral - 92% (Strongly) Agree and the “Would recommend course” question was answered 0% (Strongly) Disagree - 15% Neutral - 85% (Strongly) Agree.

One observation from the course is that Rust lends itself to revisiting a programming problem a number of times as one learns about more advanced features of the language. This is consistent with Rust’s position as a high-level systems language. As a high-level language, there is typically a simple and “good enough” means of accomplishing a task, but with some inefficiencies such as extraneous indirections or allocations. As a systems language, there is the opportunity to refine an initial implementation to obtain a more efficient solution. This arises often in the “Challenge” tasks in the programming assignments for the course, but also occasionally in the lecture material.

With regards to the Rust Education Workshop’s interest “in the use of Rust as a teaching language in courses that are not directly language related: for example operating systems, networking, or embedded programming”, I have some concerns. While I believe that most students got a lot out of the course, I would not necessarily characterize many of them as fluent Rust programmers at the end of the 15-week semester. Teaching Rust and expecting students to gain some fluency in the language as an aside to the main educational goal of the course would appear to be difficult. I would recommend structuring such courses with a significant amount of starter code for students to use, so that their programming work is focused on very particular components.

Acknowledgments

Portions of my course materials have been based on similar courses offered at Northwestern University (Jesse Tov, EECS-3/496: Systems Programming in Rust, Spring AY2017/18 [39], Spring AY2018/19 [40], and Spring AY2019/20 [41]), Stanford (Ryan Eberhardt and Armin Namavari, CS-110L: Safety in Systems Programming, Spring AY2019/20 [10]), and Georgia Tech (Taesoo Kim, CS-3210: Design Operating Systems, Spring AY2019/20 [20]).

References

- [1] Brian Anderson, Lars Bergstrom, Manish Goregaokar, Josh Matthews, Keegan McAllister, Jack Moffitt, and Simon Sapin. Engineering the Servo Web Browser Engine Using Rust. In *Proceedings of the 38th International Conference on Software Engineering Companion*, ICSE’16, page 81–89, New York, NY, USA, 2016. Association for Computing Machinery.
- [2] async-std. <https://async.rs/>.
- [3] Asynchronous Programming in Rust. <https://rust-lang.github.io/async-book/>.
- [4] Richard Bird. *Pearls of Functional Algorithm Design*. Cambridge University Press, November 2010.
- [5] Alex Chrichton. Procedural Macros vs Sliced Bread. <https://www.youtube.com/watch?v=g4SYT0c8fL0>, March 2019. RustLatam.

- [6] clap. <https://crates.io/crates/clap>.
- [7] const_generics. <https://rust-lang.github.io/rfcs/2000-const-generics.html>, May 2017. The Rust RFC Book.
- [8] crossbeam. <https://crates.io/crates/crossbeam>.
- [9] José Duarte and António Ravara. Retrofitting Typestates into Rust. In *Proceedings of the 25th Brazilian Symposium on Programming Languages*, SBLP'21, page 83–91, New York, NY, USA, 2021. Association for Computing Machinery.
- [10] Ryan Eberhardt and Armin Namavari. CS-110L: Safety in Systems Programming (Spring AY2019/20). <https://reberhardt.com/cs110l/spring-2020/>, 2020. Stanford University.
- [11] Wedson Almeida Filho and the Android Team. Rust in the Linux kernel. <https://security.googleblog.com/2021/04/rust-in-linux-kernel.html>. Google Security Blog.
- [12] Matthew Fluet. CSCI-541/641: Functional Programming and Haskell (Fall AY2018/19). <https://www.cs.rit.edu/~mtf/teaching/20181/psfp>, December 2018. Rochester Institute of Technology.
- [13] Matthew Fluet. CSCI-541/641: Functional Programming and Haskell (Fall AY2019/20). <https://www.cs.rit.edu/~mtf/teaching/20191/psfp>, December 2019. Rochester Institute of Technology.
- [14] Matthew Fluet. CSCI-541/641: Safe and Secure Systems Programming in Rust (Spring AY2020/21). <https://www.cs.rit.edu/~mtf/teaching/20205/psr>, May 2021. Rochester Institute of Technology.
- [15] Matthew Fluet. CSCI-541/641: Safe and Secure Systems Programming in Rust (Spring AY2021/22). <https://www.cs.rit.edu/~mtf/teaching/20215/psr>, May 2021. Rochester Institute of Technology.
- [16] Jeremy Gibbons and Oege De Moor, editors. *The Fun of Programming*. Red Globe Press, May 2017.
- [17] Jon Gjengset. *Rust for Rustaceans: Idiomatic Programming for Experienced Developers*. No Starch Press, December 2021.
- [18] Diane Hosfelt. Implications of Rewriting a Browser Component in Rust. <https://hacks.mozilla.org/2019/02/rewriting-a-browser-component-in-rust/>, February 2019. Mozilla Hacks.
- [19] Graham Hutton. *Programming in Haskell*. Cambridge University Press, 2nd edition, September 2016.
- [20] Taesoo Kim. CS-3210: Design Operating Systems (Spring AY2019/20). <https://tc.gts3.org/cs3210/2020/spring/index.html>, 2020. Georgia Institute of Technology.
- [21] Steve Klabnik. Rust's Journey to Async/Await. <https://www.youtube.com/watch?v=1J3NC-R3gSI>, June 2019. QCon New York.
- [22] Steve Klabnik and Carol Nichols. *The Rust Programming Language*. <https://doc.rust-lang.org/book/>.
- [23] Steve Klabnik and Carol Nichols. *The Rust Programming Language*. No Starch Press, August 2019.
- [24] Wen Kokke. Rusty Variation: Deadlock-free Sessions with Failure in Rust. In *Proceedings of the 12th Interaction and Concurrency Experience*, ICSE'19, 2019.
- [25] Stefan Lankes, Jens Breitbart, and Simon Pickartz. Exploring Rust for Unikernel Development. In *Proceedings of the 10th Workshop on Programming Languages and Operating Systems*, PLOS'19, page 8–15, New York, NY, USA, 2019. Association for Computing Machinery.
- [26] Niko Matsakis. Rayon: data parallelism in Rust. <http://smallcultfollowing.com/babysteps/blog/2015/12/18/rayon-data-parallelism-in-rust/>, December 2015.
- [27] Niko Matsakis. Rayon: Data Parallelism for Fun and Profit. https://www.youtube.com/watch?v=gof_0Ev71Aw, October 2016. Rust Belt Conference.

- [28] Niko Matsakis. Polonius: Either Borrower or Lender Be, but Responsibly. https://www.youtube.com/watch?v=_agDeiWek8w, October 2019. Rust Belt Conference.
- [29] Niko Matsakis and the Async Foundations Working Group. Brainstorming Async Rust's Shiny Future. <https://blog.rust-lang.org/2021/04/14/async-vision-doc-shiny-future.html>, April 2021. Rust Blog.
- [30] Josh Matthews. The Story of Stylo: Replacing Firefox's CSS engine with Rust. <https://www.youtube.com/watch?v=Y6SSTRr2mFU>, October 2017. Rust Belt Conference.
- [31] Miri. <https://github.com/rust-lang/miri>.
- [32] Peter Norvig. How to Write a Spelling Corrector. <http://norvig.com/spell-correct.html>, August 2016.
- [33] num-bigint. <https://crates.io/crates/num-bigint>.
- [34] parking_lot. https://crates.io/crates/parking_lot.
- [35] rayon. <https://crates.io/crates/rayon>.
- [36] The const generics project group. Const generics MVP hits beta! <https://blog.rust-lang.org/2021/02/26/const-generics-mvp-beta.html>, February 2021. Rust Blog.
- [37] The Rust Reference. <https://doc.rust-lang.org/reference/>.
- [38] The Rustonomicon. <https://doc.rust-lang.org/nomicon/>.
- [39] Jesse Tov. EECS-3/496: Systems Programming in Rust (Spring AY2017/18). <https://users.cs.northwestern.edu/~jesse/course/eecs396rust-sp18/>, 2018. Northwestern University.
- [40] Jesse Tov. EECS-3/496: Systems Programming in Rust (Spring AY2018/19). <https://users.cs.northwestern.edu/~jesse/course/eecs396rust-sp19/>, 2019. Northwestern University.
- [41] Jesse Tov. EECS-3/496: Systems Programming in Rust (Winter AY2019/20). <https://users.cs.northwestern.edu/~jesse/course/cs396rust-wi20/>, 2020. Northwestern University.
- [42] Aaron Turon. Designing futures for Rust. <http://aturon.github.io/blog/2016/09/07/futures-design/>, September 2016.
- [43] Aaron Turon. Zero-cost futures in Rust. <http://aturon.github.io/blog/2016/08/11/futures/>, August 2016.
- [44] typed-area. <https://crates.io/crates/typed-area>.

A Programming Assignments

I put a significant effort into designing interesting programming assignments for the course, as the only way to learn a programming language is to read and write programs. These programming assignments typically have a required component and conclude with some optional (and not extra-credit) “Challenges”; often, these tasks ask the students to implement some advanced functionality or to improve the efficiency of a component. These latter challenges seem particularly “Rusty”, as they emphasize Rust’s ability to start with a simple less-efficient implementation of a component that can be refined to a sophisticated more-efficient implementation. Sometimes, these challenges highlight limitations of features of Rust introduced thus far and invite students to return to them later in the course after more advanced features have been discussed.

While Rust has an excellent built-in facility for building and running tests, I have often found it difficult to organize an interesting assignment around small, independently testable functions. Especially for functions that manipulate non-trivial data structures, it can be burdensome to describe the input and output in tests. For many assignments, I have resorted to generating a large battery of tests automatically using the reference solution.

All of the programming assignments are distributed via Git repositories and include a substantial body of starter code. It is important for students to spend almost as much time reading (hopefully exemplary) Rust code

as writing. Most of the assignments make use of the excellent clap crate [6] for command-line argument parsing, which makes it easy to support different modes (tracing, statistics reporting, visualization, etc.) and/or subcommands within the same binary.

I am happy to share these with the Rust Education Workshop community and invite feedback. The section headings link to the code made available to students; please contact the author for the reference solutions.

A.1 Birch⁶

This assignment asks students to implement a mini-language called Birch. Birch is a simple stack-based language. The goal is to gain experience working with **struct**-s, **enum**-s, and **Vec**-s and managing simple object ownership.

The first major task is to implement the `FromStr` trait for `struct Prog(Command<Vec>)` to parse programs; the language has a very simple syntax that makes it relatively easy to parse. `str::split_whitespace` suffices for tokenization and the interesting aspect is to balance parentheses, which can be accomplished either through recursion or an explicit stack.

The other major task is implement the `Prog::exec` method to execute a Birch program by manipulating a `struct CmdStack(Vec<Command>)` and a `struct DataStack(Vec<DataElem>)`. As might be expected, many commands share common functionality; for example, all of the arithmetic commands must pop two numbers from the data stack and push a new number onto the data stack. This encourages writing methods for `DataStack` to capture common operations (e.g., `pop`, `pop_num`, `pop_2nums`) and to propagate errors using the `? operator`. The other interesting aspect of the basic solution is to recognize when it is necessary to `clone` to initialize or prepend onto the command stack.

Challenges The basic Birch language only supports 64-bit integers and command sequences as data elements. A simple challenge task is to use the num-bigint crate [33] for arbitrary-precision integers. Another simple challenge task is support a new kind of data element as a sequence of data elements along with Birch commands to manipulate such object data elements.

At the time that this assignment is due, closures have not yet been introduced. Another challenge task is to return to the assignment and minimize any remaining code duplication, particularly in the implementation of the arithmetic and comparison commands. (Actually, I use this challenge task in the lecture introducing closures to demonstrate how closures can be used to refactor the interpreter.)

The most interesting challenge task is to use slices to reduce allocations. The basic solution described above can lead to significant allocations due to the cloning of command sequences; a simple Fibonacci example Birch program allocates 2412541376 bytes and takes 15.18 seconds. I observe that the command stack can always be partitioned into command sequences, each of which occurs in the original Birch program; this allows us to refactor from `struct CmdStack(Vec<Command>)` (a vector of commands owned by the vector) to `struct CmdStack<'a>(Vec<&'a [Command]>)`, a vector of slices into the original Birch program. This leads to a significant reduction in allocations and execution time, 12992 bytes and 7.34 seconds, respectively.

A.2 Puzzles⁷

This assignment asks students to implement a puzzle solver that is parameterized by a puzzle trait and then use that trait and solver to solve JumplIN⁸ and Invasion of the Cow Snatchers⁹ puzzles. The goal is to gain experience working with **struct**-s and **enum**-s, **trait**-s, and managing object ownership.

The `PuzzleState` trait is to be implemented for types that represent a puzzle state:

⁶<https://git.cs.rit.edu/psr2215/mtfpsr/-/tree/main/prog02>

⁷<https://git.cs.rit.edu/psr2215/mtfpsr/-/tree/main/prog03>

⁸<https://www.smartgames.eu/uk/one-player-games/jumpin>

⁹<https://www.thinkfun.com/products/invasion-of-the-cow-snatchers>

```

pub trait PuzzleState {
    /// The type of moves for this puzzle
    type Move;
    /// Determines whether or not the puzzle state represents a solved puzzle
    fn is_goal(&self) -> bool;
    /// Enumerates all of the (legal) successor puzzle states of the current puzzle
    /// state, along with the move that leads to that successor puzzle state.
    fn next(&self) -> Vec<(Self::Move, Self)>
        where Self: Sized;
}

```

The first major task is to implement the solve method using breadth-first search (BFS) with hashing of states:

```

pub fn solve<P>(p0: P) -> Option<(Vec<P::Move>, P)>
    where P: PuzzleState + Eq + Hash + Clone, P::Move: Clone

```

The solve function returns Some((`ms`, `p`)) if the puzzle `p0` can be solved by the sequence of moves `ms` to a goal state `p`. The sequence of moves `ms` should be (one of) the shortest sequence of moves from `p0` to a goal state. As there may not be a unique goal state for a puzzle, the goal state `p` reached by the sequence of moves `ms` is returned. A BFS is used to find the shortest sequence of moves from `p0` to a goal state. A hash set or table is used to avoid redundant puzzle states (e.g., different sequences of moves may lead to the same puzzle state).

A simple solution uses HashSet (for the visited set) and VecDeque (for the BFS todo queue) and requires cloning of both puzzle states and moves. Recognizing exactly when to clone in order to satisfy the borrow checker is a good experience for students.

The other major tasks are to complete the crates that implement JumpIN' puzzles (help rabbits jump around a forest to the safety of their holes) and Invasion of the Cow Snatchers puzzles (move a UFO around a farm, beaming up cattle and avoiding obstacles).

For the JumpIN' puzzles, I provide the `struct` `JumpIN` type to represent a puzzle state, the implementation of the `FromStr` trait for `JumpIN` to parse puzzles from text files, and most of the implementation of the `PuzzleState` trait for `JumpIN`. Students are asked to complete two helper methods used by the `<JumpIN as PuzzleState>::next` method that correspond to specific kinds of moves in the puzzle.

For the Invasion of the Cow Snatchers puzzles, I provide very little, asking students to design `struct Farm { ... }` and `struct IotCS<'a> { farm: &'a Farm, ... }` types to represent the puzzle state and to implement the `FromStr` trait for `Farm` to parse puzzles from text files and to implement the `PuzzleState` trait for `IotCS`. The distinction between the `Farm` and `IotCS` types is that the former represents fixed elements of the puzzle (which differ from one puzzle to the next, but do not change when solving a particular puzzle), while the latter represents the combination of those fixed elements (as a reference to a `Farm`) and the dynamic elements of the puzzle that do change when solving a particular puzzle. While it is straightforward to represent Invasion of the Cow Snatcher puzzles without this distinction, it is a required element of the assignment to better understand the role of references in the language.

Challenges The basic implementation of `solve` requires `P: Clone` (and `P::Move: Clone`). The `Clone` trait bound is unfortunate, especially if puzzle states are “large” and expensive to copy. (Of course, if puzzle states are “large” and expensive to copy, then it may be prohibitive to keep track of all puzzle states that have been visited.) One challenge task is to eliminate the `P: Clone` and/or `P::Move: Clone` trait bounds, while maintaining an efficient solver. I observe that it would be nice to manipulate *references* to puzzle states, rather than *owned* puzzle states, but that there are difficulties with moving puzzle states into some data structure with a lifetime that lives throughout the execution of the `solve` function. This motivates another approach that uses reference-counting pointers and students are invited to return to the assignment after they have been introduced; careful thought and a well-timed `std::mem::drop` allows one to use `Rc::try_unwrap` to recover the ownership of the final goal state. I also return to this assignment during the `unsafe` lecture, because the gallery and typed-arena abstractions also suffice to eliminate the `P: Clone` trait bound.

Another challenge task to implement other movement puzzles, such as Rush Hour¹⁰ or Tip Over¹¹.

Yet another set of challenge tasks is to eliminate some of the code duplication between puzzles. For example, many puzzles use a fixed grid with a `struct Pos{x: usize, y: usize}` type in a private `pos` module that maintains the invariant that the x- and y-coordinates are within bounds. However, different puzzles use different sized grids, so it isn't immediately obvious how to share code. One option is to use `const` generics [7, 36] to implement `struct Pos<const XMAX: usize, const YMAX: usize>{x: usize, y: usize}`, where the bounds on the x- and y-coordinates are captured in the type; individual puzzles can then declare a type alias that fix the bounds appropriate for that puzzle.

The `jumpin` and `iotcs` binary crates in the starter code have significant code duplication and significant missing functionality (they simply load the puzzle from a file, solve the puzzle, and print the sequence of moves leading to a solution). An ambitious challenge task is to implement a fully-featured interactive puzzle library crate that defines an `IPuzzle` (interactive puzzle) trait and an `ipuzzle::main<IP: IPuzzle>()` function. The idea is that the `IPuzzle` trait encapsulates the requirements for an interactive puzzle and the `ipuzzle::main<IP: IPuzzle>()` function executes an interactive puzzle. Ideally, the execution of an “interactive” puzzle would be actually interactive: display the current state, check if it is a goal state and, if not, prompt for a sequence of moves and either execute the moves (if they are valid) or declare the moves invalid, and loop. Other useful functionality would include commands like `check` (to show whether or not the current state is solvable, without displaying the solution), `solve` (to display a solution), `moves` (to display the set of valid moves), `hint` (to display the initial move of a solution), and `undo` (to revert the previous move).

A.3 Tries¹²

This assignment¹³ asks students to implement a fully-featured library for a trie, which is an efficient map from strings to values, and use it to implement a statistical spelling correction program based on an idea of Peter Norvig [32].

The `struct TrieMap<V>` (provided) represents a map from strings to values:

```
pub struct TrieMap<V> {
    /// The total number of string/value pairs represented by the `TrieMap`,
    /// including this node and all descendant nodes.
    /// Note that this is *not* the total number of `TrieMap` nodes; rather, it
    /// is the total number of `TrieMap` nodes that have a `Some` `val` field.
    len: usize,
    /// The value, if any, at this node of the `TrieMap`; the corresponding
    /// string is implicit, determined by the sequence of characters from the
    /// root `TrieMap` to this node.
    val: Option<V>,
    /// The children tries, as a mapping from a character to a `TrieMap`, stored
    /// in lexicographic (i.e., dictionary) order by characters; thus, it is
    /// possible to binary search the children tries by the character.
    /// As an invariant of the `TrieMap`, no child trie should be empty.
    children: Vec<(char, TrieMap<V>)>,
}
```

The major task of the assignment is to implement `next` (return the child trie reached by a character), `get` (must use iteration and not recursion), `get_mut` (must use iteration and not recursion), `insert` (must use iteration for full credit), `remove`, `into_iter`, `iter_mut`, and `iter` methods for `TrieMap<V>`. The distinction between iteration and recursion highlights the fact that the implicit reborrowing at a recursive call can sometimes simplify lifetime management, but is typically less efficient than an iterative implementation and is susceptible to stack exhaustion. The search methods are expected to use the slice type’s `binary_search_by_key` method.

¹⁰<https://www.thinkfun.com/products/rush-hour>

¹¹<https://www.thinkfun.com/products/tipover>

¹²<https://git.cs.rit.edu/psr2215/mtfpsr/-/tree/main/prog04>

¹³Based in part on an assignment by Jesse Tov for his EECS-3/496: Systems Programming in Rust course at Northwestern University (Spring AY2017/18 [39], Spring AY2018/19 [40], and Spring AY2019/20 [41]).

In addition, students are asked to respond to two discussion prompts:

- Comment on the amount of code duplication in your implementation of the TrieMap library. Explain why (with the current features of the Rust language) it is not possible to share more code.
- Explain why the various iterators over string/value pairs return a `String` rather than a `&str`.

The former highlights the fact that Rust does not (easily) allow us to write methods that are generic with respect to whether a piece of data is owned, a mutable reference, or an immutable reference. The latter highlights the fact that the TrieMap does not actually contain the keys; rather, they must be constructed during the iteration.

The other major task of the assignment is to implement a function to compute the Damerau-Levenshtein edit distance between two strings and to use a TrieMap to efficiently enumerate the words in a corpus that are within a maximum edit distance of a word. These functions are used to demonstrate the performance difference between different methods of selecting the “best” word in a corpus to suggest for a misspelled word.

Challenges Implicit in the assignment description are a couple of challenge tasks. One is to implement `insert` and `remove` with iteration. The critical issue is updating the `len` fields of parent trie nodes when `insert` or `remove` changes the number of string/value pairs in the map. Furthermore, for `remove`, we must also delete any empty child tries from their parent’s `children` vector. One approach is to traverse the root trie twice (remembering the child indices from the first traversal in order to avoid additional binary searches), but we need to perform a reborrow in order to have two mutable references to the root for the two traversals; this does not violate Rust’s rules of references, since the original mutable reference cannot be used until the second (reborrowed) mutable reference’s lifetime has ended. Another approach, discovered by a student, is to branch and perform the reborrow at the point of iterating into a child trie of size one; this retains the ancestor trie at which the empty child trie must be pruned.

Another implicit challenge is to eliminate the code duplication in the implementation of `get` and `get_mut` and the iterators. This can be achieved by introducing code (methods and, in the case of the iterators, `structs`) that is generic over a type, which will be instantiated by either an owned type, a mutable reference type, or an immutable reference type, and has a trait bound that collects the critical operations that differ among the target code. Curiously, I have not seen this technique discussed in documentation or blogs or used in other Rust code that I have encountered; I would welcome pointers to extant descriptions of this technique. One observation is that, for `get` and `get_mut`, slightly more code is required to achieve the code sharing than was there with the code duplication; the code savings is more significant for the iterators.

Another challenge task is to minimize `String` allocations/clones for the TrieMap iterators; the ideal solution performs at most one `clone` per returned `String`. Yet another challenge task related to iterators is to make the TrieMap iterators implement the `DoubleEndedIterator` trait.

A final challenge task is to implement an entry API¹⁴ for TrieMap, which is intended to provide an efficient mechanism for manipulating the contents of a map conditionally on the presence of a key or not; the emphasis is on efficient and the methods of the entry API should not require multiple searches of the trie. Again, the `remove` method (this time for the `OccupiedEntry` type) is the tricky one. I have achieved both a safe version, which has some minor inefficiencies (one due to repeated traversals required by representing an occupied leaf node as mutable reference to the ancestor node at which it is rooted as a child trie of size one and one due to limitations of the borrow checker that motivated the introduction of the entry API) and an unsafe version, which uses a technique that is also used in the implementation of the entry API for the `std::collections::BTreeMap`.

A.4 Elementary Cellular Automata¹⁵

This assignment asks students to implement a parallel program to implement elementary cellular automata and report statistics about their execution. Conway’s Game of Life is a famous example of a cellular automaton. The elementary cellular automata are the simplest (non-trivial) cellular automata; the state is a fixed-sized one-dimensional circular grid of booleans and the next value of a cell is based on the current values of the cell and

¹⁴<https://doc.rust-lang.org/std/collections/index.html#entries>

¹⁵<https://git.cs.rit.edu/psr2215/mtfpsr/-/tree/main/prog05>

its left and right neighbors according to an update rule represented as an 8-bit value.

The task is to implement a `par::run_eca` function that executes an elementary cellular automaton (described by an update rule, a grid size, and a set of initially populated cells) with a given number of threads for a given number of steps, while recording the minimum, maximum, and final population counts of the state. Optionally, visualization may be requested, in which case the step number, visual representation of the state, and population count is displayed in each step.

The requirements are that the `par::run_eca` function must only use threading and synchronization mechanisms provided by the Rust Standard Library (and may not use any external crates). The assignment is graded based on a written description of the (intended) design, use of proper synchronization, achieving reasonable parallel speedup (with 4 threads), correctly gathering statistics, minimizing allocation/deallocation/copying, and fully supporting visualization.

The problem is “embarrassingly parallel”, in the sense that the computation to determine the next value for every cell is mutually independent. However, the work-per-cell is very small and there is some necessary synchronization at each step, which means that one must keep overheads to a minimum to observe speedup. Some non-solutions include: using a thread-pool and spawning a task for each cell; creating and destroying threads for each step of execution; putting entire grids under a `std::sync::Mutex`, copying grids between the main thread and worker threads. On the other hand, the work-per-cell is constant time, so no sophisticated load balancing is necessary.

Challenges Beyond the obvious challenge of obtaining good parallel speedup, I invite the students to implement `run_eca` functions using `rayon` [35] (a data-parallelism library), `parking_lot` [34] (provides implementations of synchronization primitives that are smaller, faster, and more flexible than those in the Rust standard library), `crossbeam` [8] (provides a set of tools for concurrent programming, including sharded reader-writer locks with fast concurrent reads and spawning threads that borrow local variables from the stack).

Implementing `run_eca` with `rayon` is, as expected, a superbly pleasant experience, nearly as simple as replacing an explicit `for`-loop iteration in the sequential implementation with a `par_iter_mut` iteration. The implementation gets satisfactory speedup, nearly as good as the direct parallel implementations.

I was not able to observe any appreciable performance advantage to using `parking_lot` or `crossbeam`.

A.5 Rock-Paper-Scissors¹⁶

In this assignment, students are asked to use `async/await` and `async-std` [2] to complete a Rock-Paper-Scissors Game Server. A game server is a good example of the need for and challenges associated with asynchronous programming. A game server should:

- handle multiple, concurrent connections from players;
- match players for individual games and referee multiple, concurrent games; and
- maintain consistent information about each player, especially with unexpected input and disconnections.

The Rock-Paper-Scissors Game Server hosts games of Rock-Paper-Scissors between concurrently connected players, while maintaining statistics (wins, losses, draws, forfeits) for each player. When a player requests to play a game of Rock-Paper-Scissors, the server attempts (with a timeout) to match them to another player concurrently requesting to play a game, at which point the players are prompted (with a timeout) to choose weapons, the outcome of the match is determined, and the players’ statistics are updated. An important aspect of the server is that once a player has requested to play a game, if an opponent is ready to play a game, then both the player’s and the opponent’s statistics must be appropriately (and atomically) updated, even if either the player or the opponent disconnects during any of the interaction up to the return to the command prompt.

A significant portion of the Rock-Paper-Scissors Game Server is provided with support for logins, changing passwords, querying online users and statistics, and quitting. Students are asked to implement functionality to support the `play` command that orchestrates the communication between players that are playing a match and

¹⁶<https://git.cs.rit.edu/psr2215/mtfpsr/-/tree/main/prog06>

properly referees the game of Rock-Paper-Scissors. In addition, students are asked to give a written description of their (intended) design. The writeup explains how first-class channel endpoints can be used to implement type-safe communication protocols (akin to session types).

Testing of asynchronous (and, therefore, non-deterministic) programs is difficult. The assignment is deployed with a test subcommand that simulates a number of concurrent players interacting with the server and monitors input and output for conformance with the specification. The testing spawns a number of players that connect to the game server; each of these players in a loop issues the play command to request a match, plays the match, and issues the stats command to compare the received statistics to the expected statistics. A percentage of the players will be (misbehaving) players; a bad player may terminate the connection at any time after issuing the play command, may timeout choosing a weapon, or may choose an invalid weapon. An additional checker player task will periodically issue the standings command and check that expected invariants hold. When the test subcommand is executed with no bad players, then it can estimate the efficiency of the game server. With cargo run --debug, the reference solution with 200 simulated players gets between 2100 games/sec and 3700 games/sec (depending on hardware and operating system), and, with cargo run --release, gets between 6700 games/sec and 17900 games/sec.

Challenges As currently implemented, the game database is lost when the server is terminated and the server is terminated by killing the process. One challenge task is to design an on-disk format for the game database and support --save <DBFILE> and --load <DBFILE> command-line arguments. Another challenge task is to have the server cleanly shut down in response to Ctrl-C.

Rust-Edu: Bringing Rust To Academia

Rust-Edu Workshop 2022
<http://rust-edu.org/workshop>



Welcome and Thank You

- Welcome to the 2022 Rust-Edu Workshop!
- Thank you
 - For being willing to bear with us as we figure the Workshop out
 - For your enthusiasm and engagement in being here
- I am so looking forward to this! Let's work together to make this thing as good as we can figure out how to



Thanks and Acknowledgements

- **Futurewei:** The vision and support of Sid Askary and the Futurewei team has made Rust-Edu possible
- **Portland State University:** Provided hosting and institutional support
- **The Rust-Edu Team:** A group of volunteers that made this all possible
- **The Workshop Program Committee:** Worked hard to make this a success
- **Cassandra Smith:** Our organizational support
- **Many Others:** Apologies to the tons of folks I've left out



Housekeeping

- Observe the Workshop [Code of Conduct](#)
- Let's keep it in-house for now
 - A recording of this meeting as well as the Proceedings will be published
 - Let me know if you have an issue with this at
workshop-f2022@rust-edu.org
- Please join the Rust-Edu Zulip [#t-workshop](#) channel!
<http://rust-edu.org/zulip>
- Leave and return as needed — there are breaks
- A bunch of Zoom stuff — next slide



Schedule

- Full schedule at <http://rust-edu.org/workshop#schedule>
- Brief outline:
 - **Session 1 - Experiences with Teaching Rust (7:30 - 9:00)**
 - **Break (9:00 - 9:15)**
 - **Session 2 - Tools for Teaching Rust (9:15 - 10:45)**
 - **Long Break (10:45 - 11:30)**
 - **Session 3 - Rust and OS (11:30 - 12:30)**
 - **Open-Ended Discussion (12:30 - 1:00)**



Zoom Stuff

- Please **mute your video and audio** unless actively speaking
- Use the Zoom Chat!
 - Post questions with **Q:** and comments with **C:** during the talk (unless the speaker requests otherwise)
 - Use **T:** in chat to speak during open discussions: the discussion moderator will call on you
 - Please keep chat discussions on-topic
- **Be good to one another.** We're all here to work together



Rust: Important To Learners

- Expressive, efficient language
- Great compile-time “guardrails” against accidents
- Rapid uptake in industry and open source
- Playground for important software ideas



Realities of PL Education

- Programming languages are considered “tech” not fundamental
 - “You will (=must) learn *many* PLs”
 - “Different tools for different jobs”
 - “Concepts, not tools”
- Lots of inertia behind teaching a few languages
- “Easy languages” are easy to teach
- “Good enough” is considered good enough
- Academia (College/University): these problems but more



Rust-Edu Wants Rust Learning To Win

- Win in academia, first and foremost
- Win in other educational settings
- If you're going to learn to program
you should learn to program in Rust



Immediate Goal: Curriculum In Academia

- We want Rust taught as a primary programming language by default in Colleges / Universities around the world
- We'd *like* Rust taught as *the* primary programming language by default in Colleges / Universities around the world
- To do that, must overcome the barriers:
 - “Just tech”
 - Inertia
 - “A hard language”
 - “Existing languages are fine”



More Work Is Needed

- Where possible, make learning tool driven
 - By enhancing existing tools (e.g. `cargo edit`)
 - By creating new tools (e.g. Rustviz)
- Improve the ecosystem for learning
 - Identify changes to Rustlang and its tools to enhance learning
 - Improve the documentation ecosystem



We've Got This

- Rust has the unique attributes to become an academic and programming learning go-to.
- It needs a big long push. You can help.
- Opportunities to change everything don't come along often.
Here's yours.



Paradigm Problems: A Case Study on rebalance

Will Crichton @ Rust-Edu Workshop – 8/16/22

CS 242: PL@ Stanford

CS 242: Programming Languages, Fall 2019

Assignments Course Policies Labs

Course summary

CS 242 explores models of computation, both old, like functional programming with the lambda calculus (circa 1930), and new, like memory-safe systems programming with Rust (circa 2010). The study of programming languages is equal parts systems and theory, looking at how a rigorous understanding of the syntax, structure, and semantics of computation enables formal reasoning about the behavior and properties of complex real-world systems. In light of today's Cambrian explosion of new programming languages, this course also seeks to provide a conceptual clarity on how to compare and contrast the multitude of programming languages, models, and paradigms in the modern programming landscape. See the schedule below for full topic list. Prerequisites: 103, 110.

Course info

- **Lectures:** Mon & Wed 4:30pm-5:50pm,
Skilling Auditorium

- **Links:** [Gradescope](#), [Campuswire](#), [feedback](#)

▷ Instructor



Will Crichton

▷ Course assistants

From Theory to Systems: A Grounded Approach to Programming Language Education

Will Crichton

Stanford University

wcrichto@cs.stanford.edu

The motivation for Rust

"For the remainder of the course, we're shifting away from formalisms of existing type theories to exploration of exciting new type theories.

We're going to look at different kinds of invariants we can encode in our type systems, like memory safety, data race avoidance, provably correct state machines, and more.

At each step, we're going to ask: what's a source of bugs in systems programming? And how can we use a type system to statically identify those bugs?"

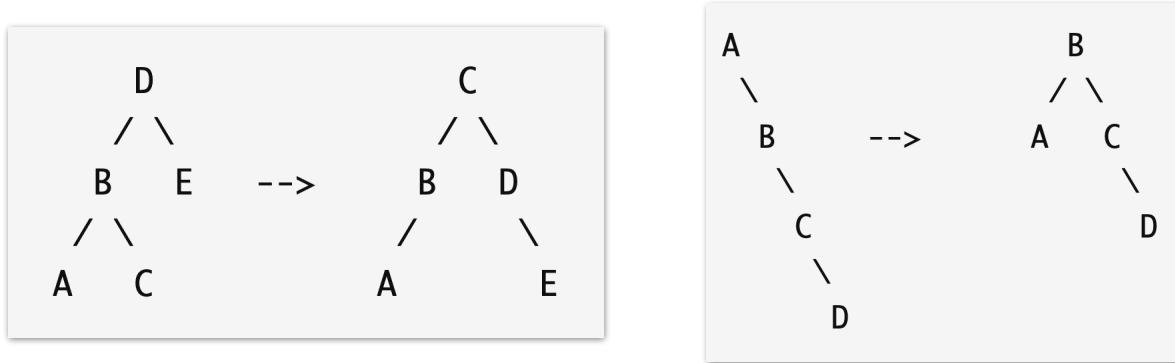
What homework problems really tested a student's knowledge of Rust?

The rebalance problem

```
enum BinaryTree<T> {
    Leaf,
    Node(T, Box<BinaryTree<T>>, Box<BinaryTree<T>>)
}

impl<T: PartialOrd> BinaryTree<T> {
    fn rebalance(&mut self) { /* ... */ }
}
```

"lift the closest element on the larger sub-tree up to the root,
rotating the former root as the root of a subtree"



Our solution

```
pub fn rebalance(&mut self) {
    if let Node(_, l, r) = self {
        let (ln, rn) = (l.len(), r.len());
        if ln > rn + 1 { If the left subtree is too large...
            let s2 = l.right_spine().unwrap(); Detach the largest left-subtree element
            let l = mem::replace(l, Box::new(Leaf)); Detach the left subtree
            let self_owned = mem::replace(self, Leaf); Detach the entire tree
            *self = Node(s2, l, Box::new(self_owned)); Reassemble the new tree
        } else if rn > ln + 1 { (same thing for the other case)
            let s2 = r.left_spine().unwrap();
            let r = mem::replace(r, Box::new(Leaf));
            let self_owned = mem::replace(self, Leaf);
            *self = Node(s2, Box::new(self_owned), r);
        }
    }
}
```

Our solution

```
fn left_spine(&mut self) -> Option<T> {
    match self {
        BinaryTree::Leaf => None,
        BinaryTree::Node(_, l, r) => match l.left_spine() {
            Some(t) => Some(t),
            None => {
                let r_owned = mem::replace(r, Box::new(BinaryTree::Leaf));
                let self_owned = mem::replace(self, *r_owned);
                match self_owned {
                    BinaryTree::Node(s, _, _) => Some(s),
                    BinaryTree::Leaf => unreachable!(),
                }
            }
        },
    }
}
```

Traverse the tree until you find the largest node (no right child)

**Student in office hours:
“I spent 8 hours trying to solve this
problem before I came here...”**

Me: 

What makes this problem hard?

- **Functional programming**
 - Inductive algebraic data type => matches and recursive function calls
 - Option types everywhere
- **(Safe) imperative programming**
 - No null pointers! Need to mem::replace **everything**.
 - T vs. &T vs. &mut T vs. Box<T> vs. &mut Box<T> vs. Option<T> vs. ...
 - Can't use clone due to generics
 - Automatic dereferencing => hard to know the type of a thing
 - (this class was also pre-Rust-Analyzer!)
 - Designing ownership-safe helper functions



Examples of poor helper function design

```
fn helper<'a>(  
    &'a self,  
    query: &T,  
    root: &'a T  
) -> Option<&'a T>
```

```
fn find_parent(  
    &self,  
    query: &T  
) -> Option<&BinaryTree<T>>
```

```
fn get_left_tree(  
    &self,  
    query: &T  
) -> Option<&Box<BinaryTree<T>>>
```

rebalance: a paradigm problem?

- Extremely hard to guess-and-check your way to a solution
 - No amount of compiler feedback will help you design a helper function
- Touches on several challenging aspects of ownership
 - Owned values vs. references, overlapping borrows, ownership+ADTs, etc.
- Can possibly serve as a litmus test for whether someone really groks the relevant parts of Rust

Copilot can't solve it either! (whew)

```
/// This function performs a single rebalancing operation on the BST in-place (if applicable).
/// The rebalancing algorithm is to lift the closest element on the larger sub-tree up to the root,
/// rotating the former root as the root of a subtree. For example:
///      D          C
///     / \        / \
///    B   E  -->  B   D
///   / \        /   \
///  A   C      A     E
pub fn rebalance(&mut self) {
    match self {
        BinaryTree::Leaf => (),
        BinaryTree::Node(v, l, r) => {
            if l.len() - r.len() > 1 {
                let (v, l, r) = self.rotate_left();
                *self = BinaryTree::Node(v, l, r);
            } else if r.len() - l.len() > 1 {
                let (v, l, r) = self.rotate_right();
                *self = BinaryTree::Node(v, l, r);
            }
        }
    }
}

error[E0308]: mismatched types
--> src/main.rs:46:11
5 |     Leaf,
|     ---- unit variant defined here
...
45 |     let (v, l, r) = match r {
|     -> this expression has type `&mut Box<BinaryTree<T>>`
46 |         BinaryTree::Leaf => panic!("Cannot rotate a leaf"),
|         ^^^^^^^^^^ expected struct `Box`, found enum `BinaryTree<T>`  

= note: expected struct `Box<BinaryTree<T>>`  

     found enum `BinaryTree<T>`
```

running 3 tests
test test::rebalance1_test ... FAILED
test test::rebalance2_test ... FAILED
test test::rebalance3_test ... FAILED

failures:

---- test::rebalance1_test stdout ----
thread 'test::rebalance1_test' panicked at 'assertion failed: `(left == right)`
left: `Node("B", Leaf, Node("C", Leaf, Leaf))`,
right: `Node("C", Node("B", Node("A", Leaf, Leaf), Leaf), Node("D", Leaf, Node("E", Leaf)))`

= note: expected struct `Box<BinaryTree<T>>`
 found enum `BinaryTree<T>`

Paradigm problems can pinpoint learning challenges

- “Ownership in Rust is hard” => “people cannot design the right type signature for a helper function when rebalancing a BST”
- Very successful strategy in CS education research
 - “Rainfall” problem has been the subject of dozens of papers! [1]
 - Small problems can tell you a lot about higher-level skills [2]
- Have you designed or encountered any paradigm problems?

[1] Seppälä et al. “Do We Know How Difficult the Rainfall Problem Is?” Koli Calling 2015. 71

[2] Stuart Reges. “The Mystery of “b := (b = false)”” SIGCSE 2008.



Experiences of Teaching Rust and Code Recommendation to Assist Rust Beginners

Hui Xu

School of Computer Science
Fudan University

8/20/2022

1



Outline

- I. Background
- II. Experiences of Teaching Rust
- III. Code Recommendation to Assist Beginners

2

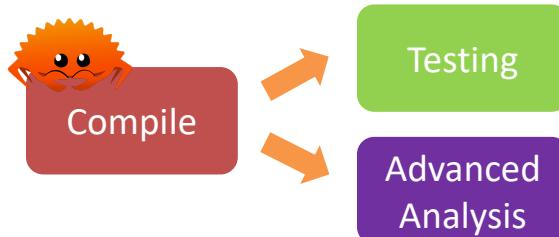
Short Bio

- ❖ Tenure-track Associate Professor, Fudan University
- ❖ Research Interest: program analysis and software reliability
 - Several publications related to Rust
- ❖ Courses I teach at Fudan:
 - COMP 737011 - Memory Safety and Programming Language Design
 - Postgraduate course
 - Design of Rust and the memory-safety issues it aims to address
 - SOFT 130061 - Compiler Theory
 - Undergraduate course
 - A few concepts related to Rust (e.g., type system)

3

Research Interest In Rust Language

- ❖ In software reliability research, we cannot trust developers.
- ❖ There are already many papers working on detecting bugs through software testing, static analysis, etc.



- ❖ Rust starts a new trial that aims to **prevent critical bugs through language design while still offering adequate control flexibility.**
 - It is challenging to balance between security and usability.
- ❖ Our research work:
 - Rust bug survey [1]
 - Rust program analysis: RULF [2], SafeDrop [3]

[1] "Memory-safety challenge considered solved? An in-depth study with all Rust CVEs", TOSEM, 2022.

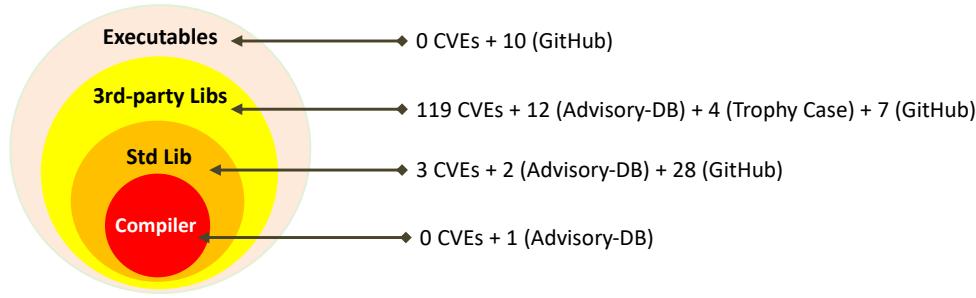
[2] "RULF: Rust library fuzzing via API dependency graph traversal." ASE, 2021.

[3] "SafeDrop: Detecting memory deallocation bugs of Rust programs via static data-flow analysis." TOSEM, 2022.

4

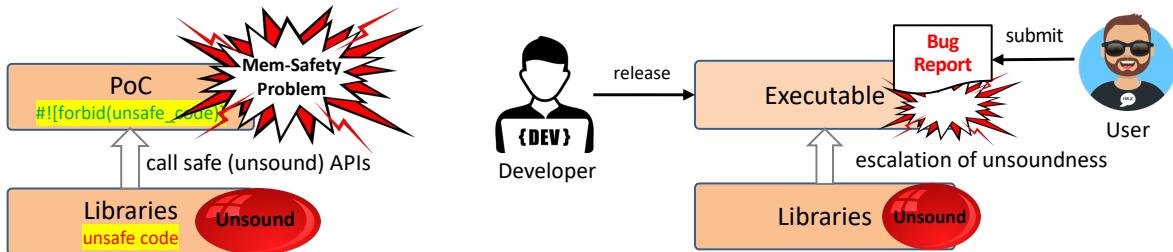
Result of Survey

- ❖ Based on a dataset of 185 memory-safety bugs before 2020-12-31



- ❖ Rust is effective in memory-safety protection:

- All these bugs require unsafe code except one compiler bug.
- Most CVEs are API soundness issues (no CVEs of executables).



[1] "Memory-safety challenge considered solved? An in-depth study with all Rust CVEs", TOSEM, 2022.

Why Do I Teach Rust?



- ❖ Rust is a successful language.
 - My student told me “As long as a Rust program compiles, the executable is likely to work correctly.”
 - One senior Rust developer said “I can always feel my skill improvement in using the language.”
- ❖ A new language with few “legacy features”
 - e.g., C++ intelligent pointers vs Rust ownership + RC
- ❖ Appealing features of Rust:
 - Memory-safety guarantee if developers do not use unsafe APIs
 - Powerful type system: type inference, generic, trait bound, etc.
 - Exception handling design: Result/Error type, unwinding/abort
 - ...

Sample Features I Like

- ❖ Variable declaration grammar: type after identifier
 - Much easier to develop an efficient top-down parser (compiler)
 - Compact for type inference: type can be omitted

Rust Code

```
let x:i32 = 1;
let y = 2;
```

C/C++ Code

```
int32_t x = 1;
auto y = 2;
```

- ❖ Trait bound: to declare bounded generic parameters

- Useful for debugging and safety control (Send/Sync)

Rust Code

```
fn max<T:Ord>(x:T,y:T)->T{
    if x > y {x} else {y}
}
```

C/C++ Code

```
template <typename T>
T max(T x, T y) {
    return (x > y) ? x : y;
}
```

7

Outline

I. Background

II. Experiences of Teaching Rust

III. Code Recommendation to Assist Beginners

8

COMP 737011 - Memory Safety and Programming Lang. Design

Part1: Foundations of Memory Safety

- Week1: Course Introduction
- Week1: Buffer Overflow
- Week2: Memory Allocation
- Week3: Heap Attack and Protection
- Week4: Memory Exhaustion and Exception Handling
- Week5: Concurrent Memory Access

❖ 3 * 45 minutes each week

- 2 units for teaching
- 1 unit for in-class practice

❖ Grading

- Attack experiment
- Coding practice
- Paper presentation

Part2: Rust Programming Language

- Week6: Rust Ownership-based Memory Management
- Week7: Rust Type System
- Week8: Rust Concurrency
- Week9: Rust Functional Programming
- Week10: Rust Compiler Design
- Week11: Rust Compiler Techniques (by Guest Speaker)

Part3: Advanced Topic for Memory Safety

Due to Covid-19, we have to rearrange the course materials.

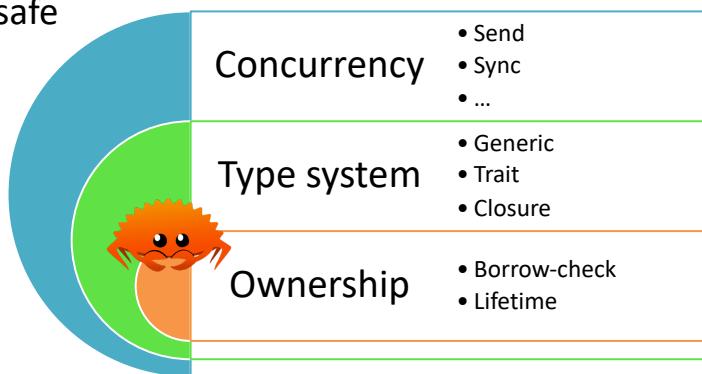
- Week12: Dynamic Analysis of Rust Programs
- Week13: Static Analysis of Rust Programs
- Week14: Presentation (by Students)

9

Course Website: <https://hxuhack.github.io/lecture/memsafe>

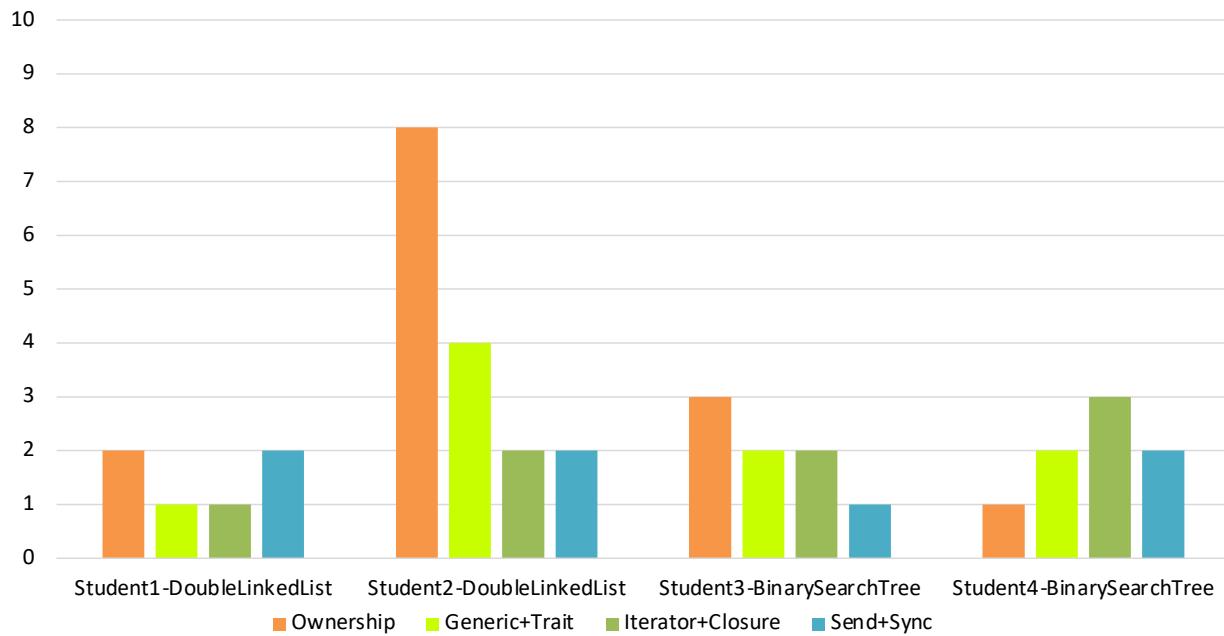
Four Rust Coding Practices

- ❖ I: Implement a binary search tree or a double linked list
 - Support insertion, deletion, and search
 - Use safe Rust only
- ❖ II: Extend the struct with generic parameters and traits
 - Support generic parameters
 - Implement traits such as Eq and Ord
- ❖ III: Implement an iterator for the struct
 - Demonstrate how the filter works with closure
 - Optional feature: collect(), map()
- ❖ IV: Rewrite the struct to be thread-safe
 - Implement Sync and Send traits
 - Show the struct is thread-safe



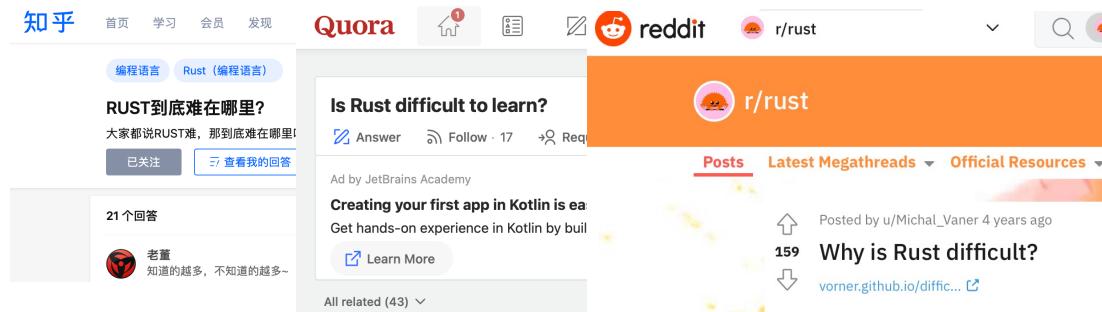
Time Spent on Each Practice

- ❖ Most students can finish the assignments in 2 hours.



11

Is Rust Difficult to Learn?



The screenshot shows a comparison between two platforms regarding the difficulty of learning Rust. On the left, a Zhihu search result for "RUST到底难在哪里?" (Where is Rust difficult?) shows a question with 21 answers and a link to "Creating your first app in Kotlin is ea...". On the right, a Quora search result for "Is Rust difficult to learn?" shows a question with 159 answers and a link to "Why is Rust difficult? vorner.github.io/diffic...". Both pages include the official Rust subreddit link: r/rust.

- ❖ Responses from my students:

- “Unfamiliar with the ownership”
- “Have much restrictions on developers”
- “Difficult but interesting. I spent much time combating with the compiler’s borrow check and dereference issues.”
- “Not that difficult if with C++ background, but I think lifetime is hard.”

12

My Understanding of Rust's Steep Learning Curve

❖ Assume a Minimal Rust for beginners? (as I tried in my class)

- Still not easy to write compilable code but should be manageable
- Exclusive mutability principle (borrow check)
- Lifetime mechanism (lifetime inference)

❖ Many advanced features

- Bring barriers to reading Rust code written by others
- Difficult to use these features well
 - e.g., Safe/unsafe, trait bound
 - C/C++ developers may ignore the soundness of their APIs

13

Outline

I. Background

II. Experiences of Teaching Rust

III. Code Recommendation to Assist Beginners

14

Code Recommendation (Our Ongoing Project)

- ❖ Build a knowledge base that summarizes the common mistakes made by Rust developers
- ❖ Make recommendations to developers when coding
- ❖ Features can be considered:
 - Compiling errors related to borrow check and lifetime
 - =>Provide better suggestions to fix the bug
 - Unnecessary usage of unsafe code
 - =>Suggest equivalent safe code
 - Other common patterns of bugs
 - =>Warn developers the problem

15

Example of Replaceable Unsafe Code: MaybeUninit

```

19     static ref BYTE_TO_EMOJI: [String; 256] = {
20     -     // SAFETY: safe
21     -     let mut m: [MaybeUninit<String>; 256] = unsafe { MaybeUninit::uninit().assume_init() };
22     +     const EMPTY_STRING: String = String::new();
23     +
24     +     let mut m = [EMPTY_STRING; 256];
25         for i in 0..=255u8 {
26             -         m[i as usize] = MaybeUninit::new(byte_to_emoji(i));
27             +         m[i as usize] = byte_to_emoji(i);
28         }
29         -         unsafe { mem::transmute::<_, [String; 256]>(m) }
30         +         m
31     };

```

16

<https://github.com/bottom-software-foundation/bottom-rs/pull/6/files>

79

Example of Replaceable Unsafe Code: Raw Pointer

```

670 -     fn read_raw_bytes(&mut self, s: &mut [MaybeUninit<u8>]) -> Result<(), String> {
671 +     fn read_raw_bytes_into(&mut self, s: &mut [u8]) -> Result<(), String> {
671         let start = self.position;
672 +         let end = start + s.len();
673 -         assert!(end <= self.data.len());
673
673
673         // SAFETY: Both `src` and `dst` point to at least `s.len()` elements:
673         // `src` points to at least `s.len()` elements by above assert, and
673         // `dst` points to `s.len()` elements by derivation from `s`.
673         unsafe {
673             let src = self.data.as_ptr().add(start);
673             let dst = s.as_mut_ptr() as *mut u8;
673             ptr::copy_nonoverlapping(src, dst, s.len());
673         }
673
673
673         self.position = end;
673
672 +         self.position += s.len();
673 +         s.copy_from_slice(&self.data[start..self.position]);
674         Ok(())
675     }

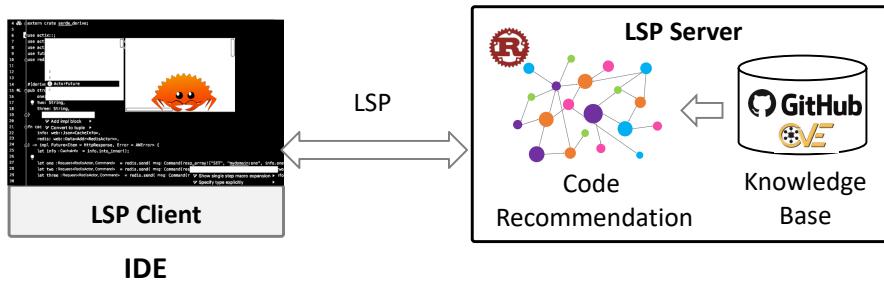
```

17

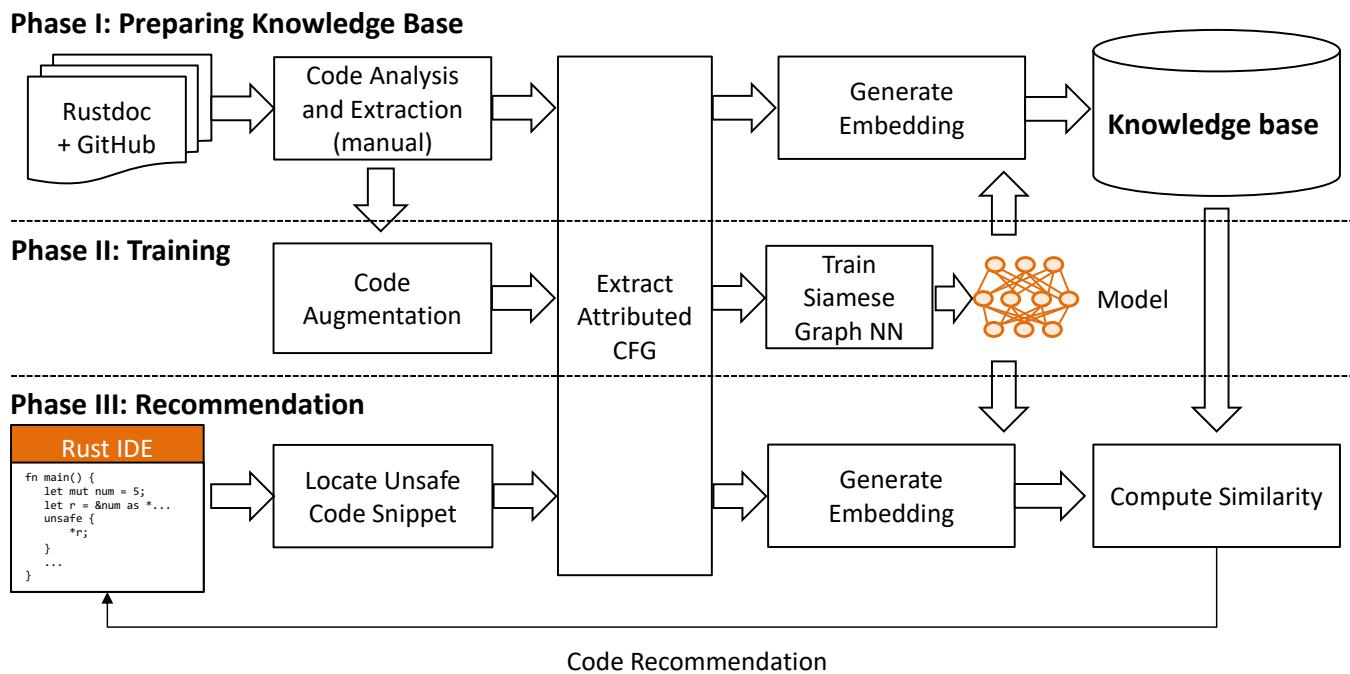
<https://github.com/rust-lang/rust/pull/83465/files>

Solution Overview

- ❖ Based on the language server protocol
 - Or Rust Analyzer (<https://rust-analyzer.github.io>)
- ❖ Advantages:
 - Rely on the power of the server to do complicated analysis tasks, e.g., static analysis, machine learning
 - Perform analysis when coding instead of when compiling
 - One server for several clients
 - Incremental knowledge base

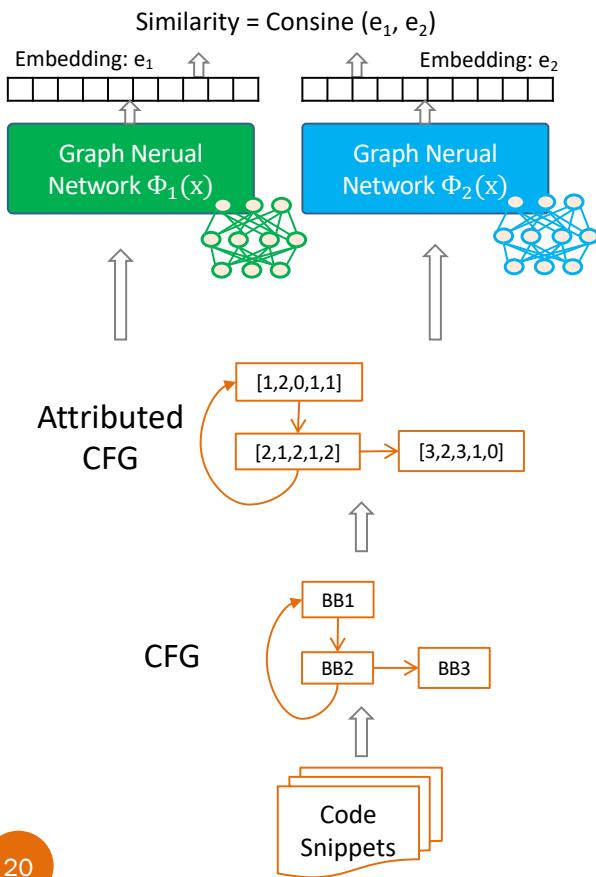


Recommendation Approach



19

Demonstration of Data Processing



❖ Siamese graph neural network

- Similarity between the same code: 1
- Similarity of different code snippets: 0

❖ Attributed control-flow graph

- A directed graph of vectors
- Each vector represents the features of a basic code block
 - Number of statements
 - Indegree
 - Outdegree
 - ...

20

Conclusion and Takeaways

- ❖ Rust is a successful language with many attractive features.
- ❖ My experiences of teaching (minimal) Rust is encouraging.
 - Positive feedback based on the performance of my students
- ❖ The magic of Rust lies in the soundness requirement of safe APIs.
 - Declarative security
- ❖ To assist Rust beginners in writing high quality code, we can summarize common bug patterns and make recommendations.
 - Language server protocol
 - Siamese graph neural network

21

Thanks for Watching

Q & A

22

82

The Book & Rustlings adaptation by JetBrains

Yet another way to learn Rust

Vitaly Bragilevsky (JetBrains)

August 20, 2022



How one can learn a programming language

- Read a book
- Look at examples
- Solve problems (with tests!)

For example, in Rust we've got:

- The Rust Programming Language:
<https://doc.rust-lang.org/book/>
- Rust By Example:
<https://doc.rust-lang.org/rust-by-example/>
- Rustlings:
<https://github.com/rust-lang/rustlings>

3

Something is missing...

Tooling!

4

Course Structure

Code

Description

Ways Variables and Data Interact: Move

Multiple variables can interact with the same data in different ways in Rust. Let's look at an example using an integer in the code snippet below.

```
let x = 5;
let y = x;
```

Assigning the integer value of variable `x` to `y`

We can probably guess what this is doing: "bind the value `5` to `x`; then make a copy of the value in `x` and bind it to `y`." We now have two variables, `x` and `y`, and both equal `5`. This is indeed what is happening, because integers are simple values with a known, fixed size, and these two `5` values are pushed onto the stack.

Now let's look at the `String` version:

```
let s1 = String::from("hello");
let s2 = s1;
```

This looks very similar to the previous code, so we might assume that the way it works would be the same: that is, the second line would make a copy of the value in `s1` and bind it to `s2`. But this isn't quite what happens.

Take a look at Figure 1 to see what is happening to `String` under the covers. A `String` is made up of three parts, shown on the left: a pointer to the memory that holds the contents of the string, a length, and a capacity. This group of data is stored on the stack. On the right is the memory on the heap that holds the contents.

Course Controls

Run Next ✓ Done

5

Who Owns the One Ring?

Sauron forged the One Ring for himself, but there were multiple owners through the times. Isildur took the ring after defeating Sauron, but then lost it. Déagol found it, but Gollum killed him and took the ring from him. See the rest of the story in the code.

We want to check the ownership at three checkpoints. Comment out the lines with `println!` to make the ownership check results correct (and the code compiled!).

Hint 1 >

Hint 2 >

Check

6

The screenshot shows the IntelliJ IDEA interface with the following details:

- Project Bar:** Shows the course structure under "course_preview > Understanding Ownership > What is ownership > Who Owns the One Ring".
- Code Editor:** Displays a Rust file `main.rs` with code related to the One Ring's ownership. A specific line `let bilbo : String = gollum;` is highlighted.
- Description Panel:**
 - Title:** Who Owns the One Ring?
 - Text:** Sauron forged the One Ring for himself, but there were multiple owners through the times. Isildur took the ring after defeating Sauron, but then lost it. Déagol found it, but Gollum killed him and took the ring from him. See the rest of the story in the code.
 - Task:** We want to check the ownership at three checkpoints. Comment out the lines with `println!` to make the ownership check results correct (and the code compiled!).
 - Hint 1:** Check
 - Status:** Compilation Failed
- Bottom Status Bar:** Shows "Cargo Check 13:14 LF UTF-8 4 spaces".

What is Rustlings by JetBrains

- Texts and examples from The Book
- Tasks with tests adopted from Rustlings
- Rust toolchain + IntelliJ IDEA or CLion + IntelliJ Rust plugin
- EduTools plugin (the platform for course development)
- Rustlings plugin (a course developed in EduTools)

Course content (ver. 12, August 2022)

- 13 chapters out of 20 from The Book covered
- 210 example/theory steps
- 108 problem steps adopted from Rustlings
- 6 IDE steps

9

We suggest the following usage scenario:

- Look at the example
- Read the corresponding book fragment
- Play with the code: edit, run, explore errors or output
- Go to the next step
- Solve task (and get back to examples and texts if necessary)

10

We are not in 1:1 correspondence to original Rustlings

- Original Rustlings is a single Cargo project, every task is one file, tests are inside this file and visible by learners.
- In our adaptation, one task is a Cargo project, so there can be many files. We can show or hide them at our discretion.
 - In most steps you can see rather `main.rs` or `lib.rs`.
 - If necessary you'll see modules and `Cargo.toml`.
 - In several tasks you are allowed to see tests.

11

Q&A

Vitaly Bragilevsky

vitaly.bragilevsky@jetbrains.com

 @VBragelevsky



12

RustViz:

Interactively Visualizing Ownership and Borrowing

Cyrus Omar

Assistant Professor

Future of Programming Lab

Computer Science and Engineering

University of Michigan

Marcelo Almeida, Grant Cole, Ke Du, Emelia Lei, Serena Li, Gabriel Luo, Shulin Pan, Yu Pan, Kai Qiu, Vishnu Reddy, Haochen Zhang, Zhe Zhang, Qiyuan Yang, Yingying Zhu
Undergraduate Students (or formerly so)
University of Michigan

See paper at VL/HCC 2022 in September. Link in Zulip!

Learning Rust's borrow checker is...

*“Learning Rust ownership is like
navigating a maze
where the walls are made of asbestos and frustration,
and the maze has no exit,
and every time you hit a dead end you get an aneurysm and die.”*

Participant in a study conducted by Coblenz et al. (ICSE 2022)

“an alien concept”, “the biggest struggle”

Participants interviewed by Shrestha et al. (ICSE 2020)

“fighting the borrow checker”

Common refrain on the web (Zeng and Crichton, CoRR, vol. abs/1901.010001, 2019)

Why so tricky?

Ownership moves, reference lifetimes, and uniqueness constraints create invisible (static) state governed by somewhat subtle rules.

The programmer has to learn to **mentally simulate** these rules and the resulting state changes to reason about read/write capabilities.

General Principle: make visible the invisible

To help learners learn to mentally simulate the language's rules, it helps to initially **do the invisible work for them** and **show them** the otherwise invisible state and events.

General Principle: make visible the invisible

To help learners learn to mentally simulate the language's rules, it helps to initially **do the invisible work for them** and **show them** the otherwise invisible state and events.

- Scaffolding as the learner develops a mental model by demonstrating the key structures and concepts that they need to be able to come up with on their own.
- Helps the learner verify that the mental model is correct (prediction -> immediate verification)

RustViz: visualizing ownership and borrowing

Demo: <https://fplab.github.io/rustviz-tutorial>

Evaluation in the classroom

EECS 490 (Programming Languages) at Michigan

- 313 students over 4 semesters
- Juniors / seniors with two semesters of C++
- Half a semester of OCaml (primarily for PL prototyping) first
- Two lectures, one discussion, and a 2 week long assignment

Learning Goals – 1

Understand ownership-based memory management in Rust

- Understand that each resource has a unique identifier called its *owner*
- Understand the different ways that ownership can be moved
- Be able to determine when resources are automatically dropped (when the owner goes out of scope)
- Understand the difference between moves and copies

Assessment

Consider the following Rust program.

```
fn id(y : String) -> String {y}
fn f(x : String) -> i32 {
    println!("Hello, {}!", x);
    let z = id(x);
    println!("Goodbye, {}!", z);
    42
}
fn main() {
    println!("Welcome!");
    {
        let a = String::from("world");
        println!("Thinking...");
        let q = f(a);
        println!("The meaning of life is: {}, q);
    }
    println!("Done.");
}
```

Between which two adjacent lines of output is the string resource that `a` initially owns dropped? Explain why by describing the events related to movement and borrowing that occur in the code above.

Assessment

Consider the following Rust program.

```
fn id(y : String) -> String {y}
fn f(x : String) -> i32 {
    println!("Hello, {}!", x);
    let z = id(x);
    println!("Goodbye, {}!", z);
    42
}
fn main() {
    println!("Welcome!");
    {
        let a = String::from("world");
        println!("Thinking...");
        let q = f(a);
        println!("The meaning of life is: {}, q);
    }
    println!("Done.");
}
```

Median Score: 91%

Median time per question: 5.43 minutes

Between which two adjacent lines of output is the string resource that `a` initially owns dropped? Explain why by describing the events related to movement and borrowing that occur in the code above.

Learning Goals - 2

Understand borrowing in Rust

- Understand the difference between immutable and mutable borrows
- Understand that borrows cannot outlive the owner
- Understand non-lexical lifetimes

Challenge: Assessing understanding of borrowing

On which line of code does the lifetime of x end?

Extra Credit: Generating Your Own Visualizations

```

/*--- BEGIN Column Specification ---*/
Owner s1;
Owner s2;
Function String::from();
Function other();
--- END Column Specification ---*/
fn f(s1 : String) { // !{ InitResourceParam(s1) }
    let s2 = String::from("hello"); // !{ Move(String::from() -> s2) }
    println!("{}:{}", s1, s2);
    other(&s1, &s2); /* !{ PassByStaticReference(s1->other()),  

                      PassByStaticReference(s2->other()) } */
}/* !{ GoOutOfScope(s1), GoOutOfScope(s2) } */

```

Extra Credit: Generating Your Own Visualizations

TABLE I
EVENTS SUPPORTED BY RUSTVIZ (EXCLUDING EVENTS RELATED TO CLOSURES AND STRUCTS)

Event	Image	Hover Messages / Description
Bind (x)		(a) x acquires ownership of a resource.
Copy (x->y)		(a) x's resource is copied. (b) Copy from x to y. (c) y is initialized by copy from x.
Copy (x->None)		(a) x's resource is copied.
Move (x->y)		(a) x's resource is moved. (b) Move from x to y. (c) y acquires ownership of a resource.
Move (x->None)		(a) x's resource is moved to the caller.
ImmutableBorrow (x->y)		(a) x's resource is immutably borrowed. (b) Immutably borrowed from x to y. (c) y immutably borrows a resource.
MutableBorrow (x->y)		(a) x's resource is mutably borrowed by y. (b) Mutably borrowed from x to y. (c) y mutably borrows a resource.
ImmutableDie (y->x)		(a) x's resource is no longer immutably borrowed. (b) Return immutably borrowed resource from y to x.
MutableDie (y->x)		(a) x's resource is no longer mutably borrowed. (b) Return mutably borrowed resource from y to x.
PassByImmutableReference (x->f)		(a) f() reads from x.
PassByMutableReference (x->f)		(a) f() reads from x.
GoOutOfScope (x)		(a) x goes out of scope.
GoOutOfScope (x)		(a) x goes out of scope. Its resource is dropped.
GoOutOfScope (x)		(a) x goes out of scope. No resource is dropped.

Extra Credit: Generating Your Own Visualizations

TABLE I

EVENTS SUPPORTED BY RUSTVIZ (EXCLUDING EVENTS RELATED TO CLOSURES AND STRUCTS)

Event	Image	Hover Messages / Description
Bind (x)		(a) x acquires ownership of a resource.
Copy (x->y)		(a) x's resource is copied. (b) Copy from x to y. (c) y is initialized by copy from x.
Copy (x->None)		(a) x's resource is copied.
Move (x->y)		(a) x's resource is moved. (b) Move from x to y. (c) y acquires ownership of a resource.
Move (x->None)		(a) x's resource is moved to the caller.
ImmutableBorrow (x->y)		(a) x's resource is immutably borrowed. (b) Immutably borrowed from x to y. (c) y immutably borrows a resource.
MutableBorrow (x->y)		(a) x's resource is mutably borrowed by y. (b) Mutably borrowed from x to y. (c) y mutably borrows a resource.
ImmutableDie (y->x)		(a) x's resource is no longer immutably borrowed. (b) Return immutably borrowed resource from y to x.
MutableDie (y->x)		(a) x's resource is no longer mutably borrowed. (b) Return mutably borrowed resource from y to x.
PassByImmutableReference (x->f)		(a) f() reads from x.
PassByMutableReference (x->f)		(a) f() reads from x.
GoOutOfScope (x)		(a) x goes out of scope.
GoOutOfScope (x)		(a) x goes out of scope. Its resource is dropped.
GoOutOfScope (x)		(a) x goes out of scope. No resource is dropped.

Median Score: 100%
Median time: 103.1 minutes

60% thought that this exercise helped them better understand Rust's semantics.

Did RustViz help?

Overall assessment: 47% (very helpful) + 46% (helpful) = 95%

+

75% of students reported frequently interacting with the visualization (also observed in a small thinkaloud study)

Did RustViz help?

“I think most of them are helpful. But sometimes I had to read them twice to fully understand.”

“I looked at every visual and traced the ownership for each variable, but I found it a bit bothersome to repeatedly look back and forth from the code. The color highlighting did help, and I did enjoy that. I found the different arrows and line segments to be a bit hard to follow.”

Next Steps

- Automatic visualization generation
 - Want to maintain customizability
 - Visualization style doesn't scale well
- Alternative visualization style: events integrated as code badges?
- Integration into more courses + public release of tutorial + more learning modules! (contact me / follow @neurocy)
 - <http://fplab.github.io/rustviz-tutorial>

Next Steps

- Automatic visualization generation
 - Want to maintain customizability
 - Visualization style doesn't scale well
- Alternative visualization style: events integrated as code badges?
- Integration into more courses + public release of tutorial + more learning modules! (contact me / follow @neurocy)
 - <http://fplab.github.io/rustviz-tutorial>

Thanks!

Pluggable Interrupt OS

Gabriel Ferrer, Professor of Computer Science
Hendrix College

Goals

- Enable OS students to write a 100% pure Rust bare-metal program
- No Rust experience prior to OS course
 - Unix Shell and utilities
 - Web server
- Encapsulate code from [Writing an OS in Rust](#) blog
- No unsafe blocks in student code
- Straightforward abstractions
 - Setup
 - VGA Buffer
 - Timer interrupt
 - Keyboard interrupt
 - Background code

Curricular Context

- All CS majors/minors take a 2-semester sequence
 - Foundations of Computer Science (Python)
 - Data Structures (Java)
 - Prerequisite for all upper-level courses, including OS
- OS taught in Rust four times thus far
 - Pluggable Interrupt OS developed in the most recent incarnation (Spring 2020)
 - Will employ again in Spring 2022

Key Background Ideas

- Bare metal programming
 - No running code beyond what the student writes
 - No OS services provided
 - Numerous library calls provided (standard Rust and otherwise)
- X86 architecture
 - Easily emulated on any platform using Qemu
- VGA Buffer
 - X86 memory-mapped character-based video buffer
- Interrupts
 - Code blocks triggered by specific hardware events
 - Timer
 - Keyboard

About unsafe

- Essential for interacting with hardware
- Our uses are “scripted”
 - Copied almost verbatim from [Writing an OS in Rust](#)
- After one semester, students should not believe themselves competent to use unsafe on their own initiative.
- Important that students are aware of the issues
 - Unsafe code discussed at length during lecture
- Lecture examples
 - Reading the key code from RAM
 - Initializing interrupts
 - Ending an interrupt

“Hello World” Example

- One interrupt handling function for clock ticks
 - Print a period on each tick.
- One interrupt handling function for keyboard events
 - Print the typed character.
- One main function
 - Defines keyboard and clock interrupt functions
 - Launches handler code

```

#![no_std]
#![no_main]

use pc_keyboard::DecodedKey;
use pluggable_interrupt_os::HandlerTable;

fn tick() {
    print!(".");
}

fn key(key: DecodedKey) {
    match key {
        DecodedKey::Unicode(character) => print!("{}", character),
        DecodedKey::RawKey(key) => print!("{:?}", key),
    }
}

#[no_mangle]
pub extern "C" fn _start() -> ! {
    HandlerTable::new()
        .keyboard(key)
        .timer(tick)
        .start()
}

```

The HandlerTable data type

```

pub struct HandlerTable {
    timer: Option<fn()>, keyboard: Option<fn(DecodedKey)>, startup: Option<fn()>, cpu_loop: fn() -> !
}

```

- Startup: sets up global data, if needed
- Timer: Handles timing interrupts
- Keyboard: Handles keyboard interrupts
- Cpu_loop: “main” code that runs in the absence of interrupts

Setting up interrupts

- Create a global variable of the HandlerTable type
- Write interrupt handlers that relay interrupts to the appropriate data fields of the global HandlerTable object
 - All unsafe code is inside the provided “background” handlers
 - No need for student code to contain unsafe blocks
- Bulk of code is by Philipp Oppermann
 - HandlerTable and bridge code is my own contribution

VGA Buffer

- Mostly by Philipp Oppermann
 - print! and println! in particular
- Added a plot() function to update columns/rows.
- Memory accesses concealed behind Buffer and Writer objects
 - Unsafe blocks again concealed from user

Cooperative Multitasking Kernel

The screenshot shows a QEMU terminal window titled "QEMU". The window has a menu bar with "Machine" and "View". The terminal displays four parallel processes (F1, F2, F3, F4) running simultaneously. Process F1 calculates Pi via a power series. Process F2 computes the average of interactively entered numbers. Process F3 performs a series of steps. Process F4 also performs a series of steps. The sidebar on the right shows the total steps taken by each process: F1 (500000), F2 (1767961), F3 (16385), and F4 (6).

```
QEMU
Machine View
F1.....F2.....
Press ENTER to start a program
Average
Pi
.
.
.
Enter tolerance:0.000001 .Enter tolerance:0.00000001 .
.3.1415946535856922 .F3.....*****
* *F1
* *500000
*Enter a number (-1 to quit):4 *
*Enter a number (-1 to quit):3 *F2
*Enter a number (-1 to quit):2 *1767961
*Enter a number (-1 to quit):1 *
*Enter a number (-1 to quit):5 *F3
*Enter a number (-1 to quit):S* *16385
*Not an int *
*Enter a number (-1 to quit):10 *F4
*Enter a number (-1 to quit): *6
*****
```

Cooperative Multitasking Kernel

- Up to four processes
 - Compute Pi via power series
 - Compute average of interactively entered numbers
- Process actions
 - Request keyboard input
 - Send output to be printed to the terminal
 - Perform one step of an algorithm
- Sidebar shows total steps performed

HandlerTable

- Startup
 - Clear the screen
- Timer
 - Update system clock count
- Keyboard
 - Save most recent keypress
- CPU_loop
 - Relay most recent keypress to kernel
 - Refresh screen when system clock updated
 - Advance currently running process one step

```
static LAST_KEY: AtomicCell<Option<DecodedKey>> = AtomicCell::new( val: None);
static TICKS: AtomicCell<usize> = AtomicCell::new( val: 0);

fn tick() { TICKS.fetch_add( val: 1); }

fn key(key: DecodedKey) { LAST_KEY.swap( val: Some(key)); }

fn cpu_loop() -> ! {
    let mut kernel : Kernel = Kernel::new();
    let mut last_tick : usize = 0;
    kernel.draw();
    loop {
        if let Some(key : DecodedKey) = LAST_KEY.load() {
            LAST_KEY.store( val: None);
            kernel.key(key);
        }
        let current_tick : usize = TICKS.load();
        if current_tick > last_tick {
            last_tick = current_tick;
            kernel.draw_proc_status();
        }
        kernel.run_one_instruction();
    }
}
```

Implementation Details

- User selects current window with function keys
 - Current window highlighted with asterisks
 - Other windows surrounded with periods
- Currently selected window either:
 - Is blocked on input and receives next input, or:
 - Executes one algorithmic step

Other assignments

- VGA Video Game
 - “Ghost Hunter” example
- Video Game Kernel
 - Select/pause video game on console

Future Work

- Potential extensions:
 - Add RAM disk or disk simulator
 - Incorporate preemptive multitasking
 - Develop more model assignments
 - Handle additional interrupts
- Seeking collaborators!
- URLs:
 - https://github.com/qjf2a/pluggable_interrupt_os
 - https://github.com/qjf2a/pluggable_interrupt_template
 - https://github.com/qjf2a/ghost_hunter
 - <https://hendrix-cs.github.io/csci320>
 - https://hendrix-cs.github.io/csci320/projects/coop_os
 - <https://hendrix-cs.github.io/csci320/projects/baremetalgame>
 - https://hendrix-cs.github.io/csci320/projects/game_kernel