Imagining Introductory Rust

Conference Paper · August 2022

CITATIONS
0

READS
59

1 author:

Rose Bohrer
Worcester Polytechnic Institute
28 PUBLICATIONS **360** CITATIONS

SEE PROFILE

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

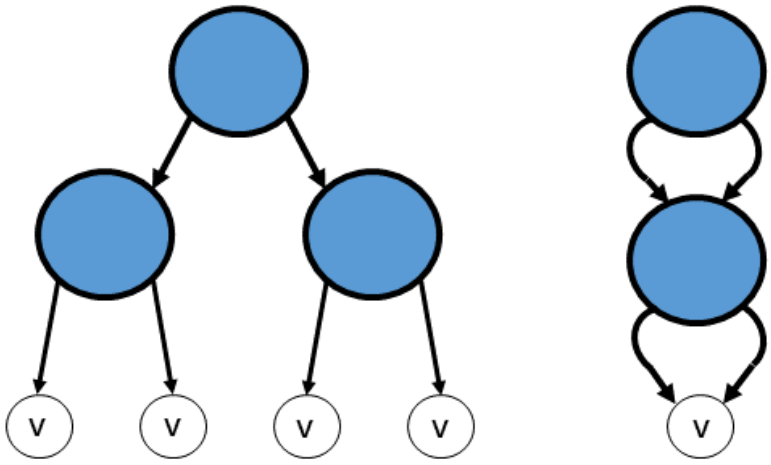

Figure 1: Tree-shaped data structure, both with aliasing (left) and without aliasing (right).

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 it 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.