RustLIVE: Reducing the Learning Barriers of Rust Through Visualization

Diane Beightol Stephens
University of Georgia
Athens, Georgia, USA
diane.stephens@uga.edu

Kyu Hyung Lee
University of Georgia
Athens, Georgia USA
kyuhlee@uga.edu

Mustakimur Rahman Khandaker
University of Georgia
Athens, Georgia USA
mrkhandaker@uga.edu

***Abstract*—This innovative practice full paper introduces an independent learning tool for the Rust programming language. Secure software begins with secure memory management as over 65% of software vulnerabilities in modern code bases are the result of memory safety problems. Rust is a promising memory safe language that is rapidly gaining traction as a replacement for C and C++ in systems software. However, Rust is challenging due to its novel approach to memory safety. The Rust programming paradigm is focused on the principles of ownership and borrowing. These concepts effectively achieve memory and thread safety but are enforced at compile-time, enabling runtime performance comparable to C but posing significant implementation challenges for students and even experienced programmers.**

We propose an innovative visualization tool, RustLIVE, that clearly depicts the most difficult concepts of the Rust language. RustLIVE is an extension for VSCode, the IDE of choice for over 60% of Rust developers. Seamlessly integrated with the Rust compiler and its borrow checker, RustLIVE requires no code annotations. Instead, it extracts necessary information directly from the compiler to provide color-coded visual timelines that illustrate the ownership of memory resources and the liveness of borrows. RustLIVE is an innovative, independent learning tool that helps learners form a correct mental model, avoid `unsafe` code and improve student experience.

***Index Terms*—computer science education, Rust, memory safety, memory safe language**

I. Introduction

Memory unsafety has been the Achilles' heel of the software industry for over a decade. Both Microsoft [1] and Google Chrome [2] among others estimate that around 70% of the security bugs in its code are due to memory errors. This struggle is realized in Computer Science education; recent research confirms that Computer Science students lack the key fundamental skills to write secure programs [3].

Secure software begins with safe memory management. Memory safety, a property of some programming languages, protects against common memory-related errors like memory leaks, dangling pointers and unchecked bounds. These errors can cause program crashes and open doors for serious attacks [4]. For instance, if a program deletes a list but later tries to read from it, what happens? A memory unsafe language may still allow access to the now-freed memory. In contrast, memory-safe languages prevent this type of access, preventing potential security breaches.

The problem stems from the use of systems languages such as C and C++ which allow the fine-grained control required for low level software but put the burden on the programmer to perform the needed protections on memory resources. C has been the language of choice for development of operating systems and embedded software because of its efficiency and performance. If you use a vending machine, a smart phone or any other smart appliance, all are likely programmed in C. In C, the programmer is responsible, and managing memory in C is difficult and a primary reason that memory vulnerabilities exist and continue to be the source of most security vulnerabilities. Yet, in a survey of Computer Science students, we found that most students displayed little knowledge or had misunderstanding about memory management [3] .

Memory safety is handled differently in different programming languages:

- **C** and **C++** allow manual memory management. The programmer is responsible for memory safety.
- **Java, Python, Go** and most other 'memory safe' languages employ a garbage collector which runs periodically to manage memory automatically, abstracting away the allocation and freeing of memory.
- In **Rust**, the compiler verifies memory safety. Code that passes the Rust compiler checks is generally memory safe.

Garbage collectors can be effective at keeping memory *tidy* but they come at a cost - programs use more memory and run slower. So, while Java and Python are useful for many applications, the runtime performance makes them less suitable for systems and embedded systems development, where performance and memory management are critical.

Government leaders are turning to memory safe language. "The highest leveraged method manufacturers can use to reduce memory safety vulnerabilities is to secure one of the building blocks of cyberspace: the programming language."Using memory safe programming languages can

eliminate most memory safety errors. “The overarching software community across the private sector, academia, and government have begun initiatives to drive the culture of software development towards utilizing memory safe languages.” [5] Furthermore, a recent gathering sponsored by the United States government, the National Science Foundation (NSF) and the National Institute for Standards and Technology (NIST) brought together leaders from academia, industry and government to identify ways to improve the security of the open-source software ecosystem. “We need to incentivize developers to write secure code and make security a part of professional training and university course curricula,”adding that the Rust programming language, despite its initial learning curve, is particularly well-suited for safe system development [6].

What if we could eliminate an entire class of the most severe memory vulnerabilities before they ever happened? That’s the approach of Rust. Rust is a memory-safe language built for systems software development [7], a replacement for C. The difference in Rust and other memory safe languages - Rust achieves memory safety without compromising performance; memory checks are performed at compile time so there is essentially zero cost to runtime performance. The Rust compiler enforces its memory safety guarantees. The core of the Rust model is its principles of *ownership* and *lifetime* of references (*borrows*).

The three tenants of Rust ownership are 1) each value has an owner, 2) there can only be one owner at a time, and 3) when the owner goes out of scope, the value will be dropped. In Rust, every resource has a single owner at any time. Ownership can be transferred, but the resource always has exactly one owner until it is no longer needed and then freed or *dropped*. This ownership model prevents data races as no two threads can simultaneously own the same resource. All resources are owned in Rust and dropped when no longer in use.

Rust allows references to a value, called *borrows*, during the lifetime of the value. Borrows allow for the sharing of resources without transferring ownership, adhering to the following rules:

- Multiple immutable borrows, aliases, are allowed.
- Only one mutable borrow at any time is allowed.
- References cannot outlive the owner variable.

The core safety principles of Rust - ownership and borrowing - are conceptually clear and straightforward. Yet, their implementation can be counterintuitive and diverge significantly from approaches in other programming languages, and without visual clues in the Rust source code. Consequently, Rust presents a steep learning curve, described as “near vertical” by one programmer [8]. Rust programmers must adopt a fundamentally different way of thinking. However, grasping these safety principles and their enforcement is pivotal for computer science students to understand safe memory management.

```
fn main() {
    let mut n_grade: i32 = entered_grade;
    let mut grades = vec![100, 90, 80, 70, 60];
    update_grades(n_grade, grades);
    println!("Grade calculated is: {}", n_grade);
    println!("Final grades: {:?}", grades);
}
fn update_grades<T>(grade:T, grades:Vec<T>) {}
```

Fig. 1. Rust program that will not compile due to the enforcement of ownership principle. The ownership of `n_grades` is transferred to `update_grades` yet ownership of `grades` remains in `main`.

The code sample in Fig. 1 will not compile due to Rust’s ownership rules. Both variables, `grade` and `grades`, are initially owned by `main`. However, ownership of `grades` is transferred to `update_grades`, while `main` retains ownership of `grade`. The source code does not visually indicate these ownership dynamics.

In this paper, we present a novel visualization tool for Rust programmers, RustLIVE. RustLIVE works where Rust programmers are - in the IDE - and provides a visual timeline of each variable to clearly depict who owns each resource and where each borrow is *live*. With this tool, Rust programmers can more readily retrain how they program - with memory safety in mind. While Rust effectively mitigates the most serious coding errors [1], [2], it introduces a paradigm that challenges programmers and students learning the language. Rust strictly enforces the safety of owned and borrowed values, and failure to adhere to these rules results in compilation errors. To aid users in navigating these requirements, RustLIVE provides visuals that clearly depict the ownership of resources and the liveness of references.

II. Related Work

Visualization has long been recognized as a valuable tool for explaining complex concepts in Computer Science education, as evidenced in previous studies [9]–[13]. The potential of visualization as a learning aid in programming education became apparent with the emergence of popular tools like Python Tutor [14], a web-based program visualization tool for the Python language. Python tutor allows students to write Python programs and step through their execution while observing the run-time state of the program’s data structures. Many universities have integrated Python Tutor into their CS1 courses. Although RustLIVE visualizes static information rather than runtime behavior, the success of Python Tutor and other visualization tools [13] has inspired educators to explore the effective use of visualization in helping budding programmers build mental models.

RustViz [15] is an existing visualization of Rust ownership and lifetimes. However, while RustViz generates detailed visualizations, it does not do so automatically but rather requires an expert (presumably teacher) to annotate the Rust source code to be visualized. Automatic

visualization is mentioned as a possible future work of RustViz.

VRLifetime [16] uses visualization in an IDE to help detect two common kinds of concurrency bugs: double lock and locks in conflicting order. Other works have tackled the issue of explaining compiler errors such as [17] and [18] that depict a directed graph representation of the lifetime constraints that lead to a lifetime error in Rust. Rust life follows the lifetime constraints violated in order to provide an informative explanation or a graphical explanation of the error.

III. Motivation

A. Compiler Enforcement of Memory Safety

A recent empirical study analyzing 100 Stack Overflow questions confirmed that programmers find Rust's memory safety policies challenging to implement [19]. This study identified that understanding the liveness of references and reasoning about the movement of owned values are particularly difficult aspects of Rust. Furthermore, discussions among Rust developers reveal widespread complaints about the complexity of the borrow checker [20]. Experienced programmers, including those familiar with C++, have noted that the borrow checker requires a significant shift in thinking. Some describe it as an "alien concept," [21] and it is common for programmers to discuss "fighting with the borrow checker." [22]

Research also highlights that a common cause of blocking bugs in Rust is the misunderstanding of lifetime rules, suggesting that even seasoned developers struggle with these complex concepts [23]. The study proposes that tools to visualize borrow lifetimes during coding could greatly assist programmers in avoiding memory bugs. Industry professionals acknowledge that Rust's steep learning curve poses a significant barrier to its adoption, further emphasizing the challenges posed by these safety features [8].

B. Visualization for Software Usability

In his report on the usability of Rust, Will Crighton [24] suggested that "tools for visualizing static information such as types and lifetimes can help programmers build a mental model of how the type checker works and reason about why individual errors occur. Displaying this information in a succinct, non-intrusive, yet informative way is still an open problem." A recent study on Usability and HCI of Rust [25], suggests visualization of Rust's lifetimes as a next step for the usability of Rust.

The need for visualization has been a topic of discussion among Rust programmers in blogs and at RustConf since at least 2016. Several blog posts propose various visualizations. One depicts a vertical timeline similar to RustLIVE [26]. A different mock-up in another blog post shows a possible approach for visualizing Rust code in an editor [27]. On the Rust Internals Forum [28], a contributor (Nashenas88) started a thread to discuss ideas for visualizing ownership and borrowing in an editor. The thread contains various ideas for such a visualization from Faria and others. One contributor was able to create a prototype in Atom. In all cases, the mockups were not substantially developed or evaluated.

Computer Science education scholars have long advocated for the use of visualization as an educational tool for understanding complex concepts. "Visualization can make focal and explicit processes that are hidden and implicit." [13] Visualization enhances comprehension of inherently abstract software structures [9]. To underscore the motivation of RustLIVE, visualization provides a conceptual model that aids learners in constructing programming knowledge [12]. This aligns with the sentiment that visualization facilitates the development of reasoning skills crucial in introductory programming education. The goal of software visualization is to improve our understanding of inherently invisible and intangible software [9]. Some scholars suggest that visualization "can illuminate critical aspects of pivotal programming concepts, aiding learners in the formation of mental representations essential for mastering programming." [11]

C. Goal of RustLIVE

RustLIVE aims to improve the usability of Rust by providing a visual timeline that clearly illustrates its most difficult concepts: the ownership of values and the liveness of references. The tool is designed to help programmers form a clear mental model of Rust's memory safety principles and to enhance learning experiences for students. Furthermore, RustLIVE integrates with the Rust compiler, functioning as an independent educational tool. This feature is particularly useful as it provides personalized feedback to students, a significant benefit in both large classroom settings and independent study environments.

IV. RustLive: Design and Implementation

Users can activate RustLIVE on demand within VS Code. This action triggers the nightly compiler, which enables us to replace the standard borrow checker with our custom version. Our version safely stores the results, allowing RustLIVE to analyze the Mid-level Intermediate Representation (MIR) and borrow computations. It gathers essential data on ownership and liveness, and delivers this information to the VS Code extension in JSON format. The RustLIVE extension then processes this JSON to produce annotated code and a timeline graph, which are displayed within the VS Code webview panel. See Fig. 2 for an illustration of the RustLIVE design.

A. Frontend - IDE Integration

The preferred development environment for Rust developers, as indicated by a recent survey [29], [30], is

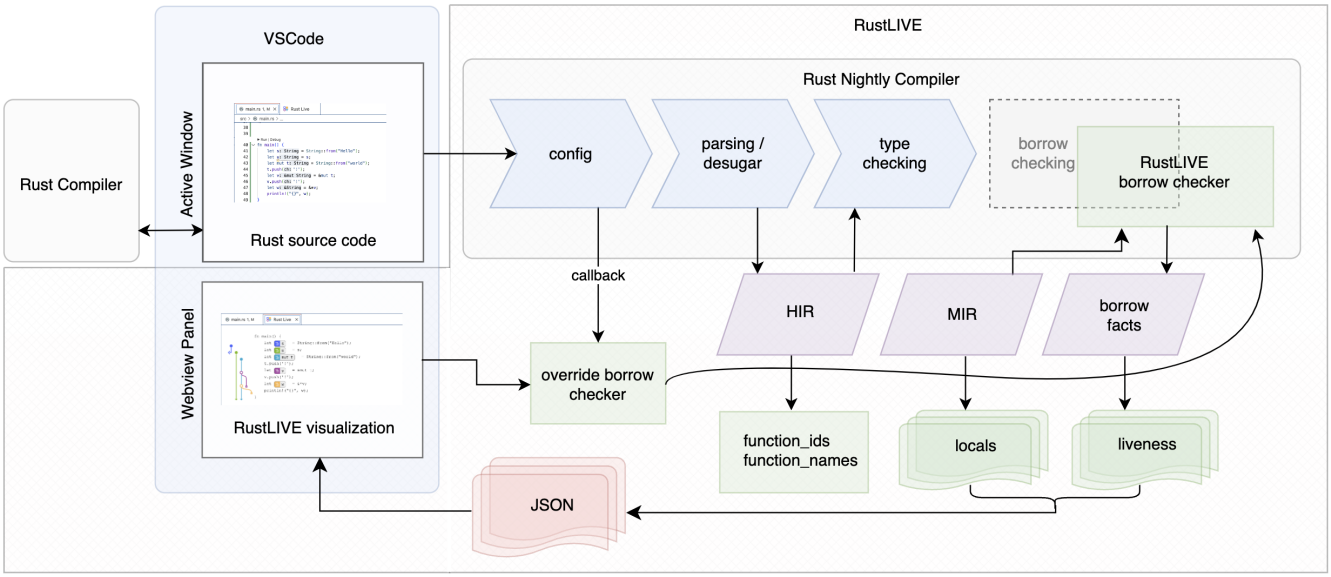


Fig. 2. Implementation of RustLIVE.

VS Code. VS Code provides an integrated environment for language development tools, along with a webview panel API that facilitates a seamless environment for Rust developers. An innovative design, RustLIVE is integrated with the Rust compiler - the enforcer of Rust's novel memory safety tenants. Specifically, the webview frontend awaits data from the compiler-query backend to generate the webview panel. The data is exported as JSON format, with each function represented by an object that contains key value pairs for each local and its associated data. The webview renders a table on the right, presenting annotated source code extracted from the active code window. On the left side, SVG graphics are dynamically created to represent the liveness of variables as a graph. This dual presentation provides programmers a visual depiction of the dynamics of ownership and lifetime of borrows as they change in the source code. The y-coordinates of the graph align with the source code lines in the table.

The JSON data includes the line number and position of each variable definition. This data enables the annotation of each variable in the source code table with a color code to match its respective liveness timeline. Owned variables are represented by solid points at the beginning of their lifetimes, while borrowed variables are depicted with hollow points. For borrowed variables referencing owned locals, the timeline starts from the referent. Liveness is indicated by a vertical line corresponding to the source lines where live. See Fig. 3.

B. Backend - Compiler Integration

Rust memory safety is enforced by the compiler, so RustLIVE interacts directly with the compiler to gather data for visualization. Specifically, RustLIVE parses the

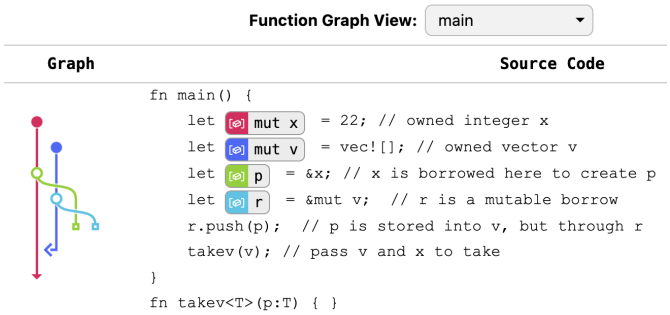


Fig. 3. RustLIVE webview panel. Owned locals x and y are solid points. Borrows r and s are hollow and originate at the referent.

MIR to obtain data on owned values. The borrow checker is responsible for verifying lifetime constraints of references and enforcing non-lexical lifetimes. MIR and the borrow checker are central components in the design and functionality of RustLIVE.

1) Rust Compiler: The Rust compiler works incrementally, progressing through several phases to transform Rust source code into executable machine code. This process involves constructing intermediate representations at different states, as illustrated in Fig. 4. Rust source code is tokenized, forming a token stream that is then parsed into an Abstract Syntax Tree (AST) [31]. The AST serves as the basis for generating the High-level Intermediate Representation (HIR), where source code is broken down into its constituent elements, macros are expanded and loops are desugared. The HIR represents the hierarchical syntax structure, grammar, of the source code, which is useful for parsing and static code analysis., HIR is lowered to MIR.

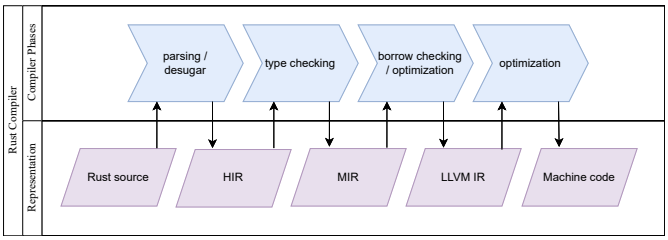


Fig. 4. Phases and intermediate representations of the Rust compile process

```rust
fn main() {
    let n_grade: i32 = 100;
    let grades: Vec<i32> = vec![100, 90, 80, 70, 60];
    {
        let _new_grade: i32 = n_grade;
        let _new_grades: Vec<i32> = grades;
    }
    println!("Grade calculated is: {}", n_grade);
    // println!("Final grades: {:?}", grades);  <-- error
}
```

Fig. 5. Ownership Transfer and Drop

Several optimization passes are performed on the MIR but, most importantly, MIR is used for borrow checking. MIR enables more flexible borrowing with better precision which allows more programs to compile and paves the way for the implementation of non-lexical lifetimes (NLL). MIR is converted to LLVM Intermediate Representation (LLVM IR) which is translated into machine code.

Why MIR?: The incorporation of the MIR intermediate representation to the Rust compiler is essential to the RustLIVE project. Internally to the compiler, MIR is a set of data structures that encode a control-flow graph (CFG). The CFG representation captures the dynamic aspect of the code, how execution moves from one statement to the next or one block to the next. This is essential to track points in the flow where variables are defined, moved, and dropped, where they are live or where they hold a value that might be used in the future. The MIR allows tracking variables as the program runs, enabling depiction of the runtime `liveness` of variables.

Transfer of Ownership: In Rust, every value has an owner and there can only be one owner at a time. This enforcement of the one-to-one relationship between owner to value sometimes requires ownership to be transferred rather than copied, especially when copying the resource is not feasible. This distinction between copying some values and transferring (or 'moving' as known in Rust) others differs from other programming languages. Moreover, in Rust, there are no visual clues to indicate whether a value is being copied or moved, which can make reasoning about ownership challenging. Consider the example program in Fig. 5. In this program, `n_grade` and `grades` are created and then assigned to new variables. We can print `n_grade` but get a compiler error if we try to print `grades`. For programmers new to Rust, this behavior is difficult to reason.

Borrow Checker: The borrow rules of the Rust type system are enforced by the borrow checker, which operates on the MIR as one of the final phases of the compilation process. In 2018, a reformulation of the borrow checker took place, resulting in the redesign and renaming to 'Polonius' as announced in a blog post by Niko Matsakis [32]. A key to the Polonius design is the enforcement of non-lexical lifetimes (NLL).

The borrow checker traverses the MIR, creates a set of numbered regions and relations based on the Rust type system rules. These relations become input facts to the borrow checker. The borrow checker determines the specific points in the MIR where regions must be live. A region is deemed live at a point if its current value may be used later. These results are collected and stored in the output facts.

2) RustLIVE integration: With the RustLIVE extension installed in VS Code, RustLIVE operates as an independent development tool for Rust programmers. No code markup is necessary as RustLIVE extracts data directly from the Rust compiler.

Compiler Callbacks: The RustLIVE compiler-query uses the `rustc_driver` interface to run the compiler. `rustc_driver` provides callbacks that are executed when specific stages of the compilation process are completed. Four callbacks provided by `rustc_driver` are:

- `config`
- `after_crate_root_parsing`
- `after_expansion`
- `after_analysis`

For RustLIVE to depict the Rust safety tenants of ownership and the liveness of borrows, it relies on accessing the MIR representation and the borrow checker facts, which are not computed until late in the compilation process. Consequently, the `after_analysis` callback is our vehicle.

The MIR intermediate representation is the resource for the ownership and borrow lifetime information useful to RustLIVE. However, MIR is computed per function in the Rust crate so we use the high-level intermediate representation, HIR, to get the function IDs, `LocalDefIds`. Specifically, we query the 'items' of the HIR abstract syntax tree and filter those that represent functions. These `LocalDefIds` provide the keys to retrieve the function names and the associated MIR representation (body) of each function. Then we begin analyzing each MIR function body.

Function Analysis: The `locals` within the MIR not only represent variables from the source code but also include temporary locations generated by the compiler. To focus solely on tracking user variables, we exclude the compiler-generated temporaries. Leveraging the information provided in the function body, specifically from `mir::Body`, including `basic_blocks`, `var_debug_info`,

`local_decls`, and `span` we obtain useful information on each `local`. This includes details such as its name and where it is defined.

By iterating through the basic blocks and statements of the MIR, we detect semantic changes such as moves (ownership transfer) and identify instances of storage live and storage dead. The MIR intermediate representation has a vastly different structure than the Rust source code so the locations must be correlated to lines in the Rust source code. For each function body (MIR), we create a map of the MIR location (basic block index, statement index) to a Rust source code line.

Borrow Checker: The most important information for RustLIVE, the liveness of borrows, could not be obtained from `mir::Body` but rather needed the input and output facts computed by the borrow checker. The borrow checker is responsible for identifying which `locals` are references, determining the owned variables they might refer to, and determining where the references are live.

The borrow checker operates with a finer location granularity within the MIR body, which we refer to as rich locations. Rich locations differentiate between the start and intermediate points of each basic block statement. Each rich location is identified by an index. The `location_table` associates each location index to its corresponding rich location.

The borrow checker computes an origin for each reference. Each origin is represented by a numbered region variable [32]. The relation created between origin to reference is referred to as a `loan`. `Origins` represent where loans are issued. The borrow checker walks the MIR and builds the set of relations that must hold according to the constraints implied by the loans. The sets are accumulated at each point, rich location, in the MIR. The relations accumulate and form input facts of the borrow checker.

The borrow checker also computes a set of output facts that includes the liveness information of regions. For every rich location, the borrow checker identifies all regions that are live at that point given the constraints. We then collect these regions for each point and utilize the `location_table` to translate from an index to a rich location and subsequently to an MIR (basic block, statement) location. Using this set of regions and MIR locations, we create the `region_map` hash map, which maps each numbered region to a vector of live MIR locations. We employ a Rust compiler method to compute a hash map of each local to the numbered region, `place_regions`, and then identify which user locals are references (borrows) and create a map from each user local referencing a borrow to its set of live locations.

C. RustLIVE Example Visualizations

1) ownership: The example program in Fig. 5 gave no indication why `grades` could not be accessed in the println! macro. However, we can see in the RustLIVE visual in Fig. 6 that the ownership of `grades` is moved

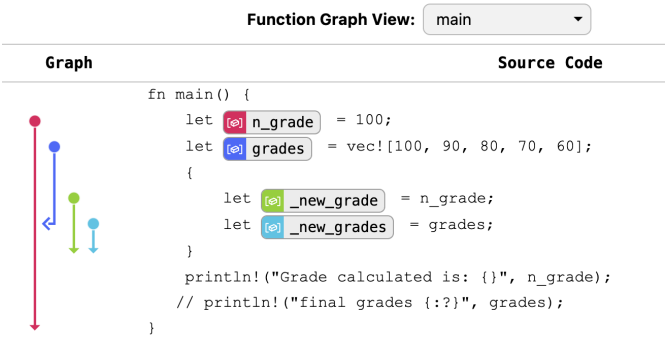


Fig. 6. Ownership Transfer and Drop Depicted in RustLIVE

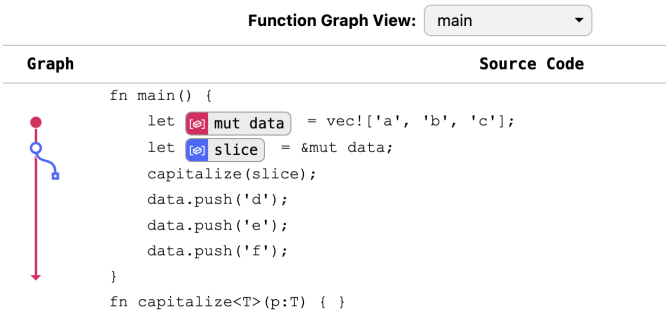


Fig. 7. RustLIVE depiction of non-lexical lifetimes (NLL)

in the inner block to enforce the only one owner principle, `grades` transferred ownership of the resource to `_new_grades`.

2) Non-Lexical Lifetimes: In the RustLIVE graph of Fig. 7, the vector `data` is borrowed and the resulting reference is assigned to a variable `slice`. `slice` is passed to `capitalize`. According to Rust safety rules, the vector can not be mutated while the reference is live. A programmer would likely reason that the lifetime of `slice` extends to the end of scope - `}` and prior to the incorporation of NLL, it did. However, with NLL we clearly see in the graph that the lifetime of `slice` ends at the call to `capitalize` so mutating the `data` vector with `data.push` is allowed.

V. Evaluation

We assessed the effectiveness of RustLIVE among the following three groups:

- **[GRP1]** Rust Workshop Participants - Individuals who attended a Rust workshop at a security conference, and are either currently using Rust in their professional roles or plan to use it in the near future.
- **[GRP2]** Graduate Students - Computer Science Masters and PhD students who have completed a graduate-level secure programming course taught using Rust.
- **[GRP3]** Undergraduate Students - Third and fourth-year undergraduate students in Computer Science who have not received formal instruction in Rust, but have

expressed an interest in secure coding and learning the Rust language.

Prior to the tool demonstration and evaluation, the undergraduates were required to watch a video explaining memory safety and the Rust model. Evaluators first examined sample Rust programs in the active code window of VS Code, and, then observed the code and visualization in the RustLIVE panel. The Rust programs used for evaluation are shown in Table II.

We requested the evaluators to provide numerical scores in response to the following questions, using a scale of one to ten, where one represents the lowest and ten represents the highest rating.

- **Q1.** How would you rate your understanding of the Rust language and how it achieves memory safety?
- **Q2.** Is the visualization provided by the tool helpful in understanding how ownership is enforced in Rust?
- **Q3.** Is the visualization provided by the tool helpful in understanding the lifetime of borrows in Rust?
- **Q4.** Overall, is the tool useful in understanding memory safety in the Rust language?
- **Q5.** Would this tool be useful in developing your Rust programming skills?

We present the evaluation results in three tables. Table I displays the overall results from all respondents. Table III presents the results from evaluators with prior Rust knowledge or experience (GRP 1 and 2). Table IV details the results from undergraduate students without prior Rust experience (GRP 3). Note that, the responses to Q1 on the understanding level of Rust differ between the groups: for GRP 3, Q1 is based on their understanding of the video regarding memory safety and the Rust model; for Groups 1 and 2, it reflects their overall understanding of the Rust language and memory safety.

With the primary objectives of RustLIVE to aid programmers in developing a mental model of the Rust ownership and lifetime principles, evaluation results confirmed the tool's effectiveness. Specifically, evaluators rated RustLIVE highly, at 8.4316 for its effectiveness in understanding memory safety in the Rust language (Q4). On examination of the effectiveness toward understanding ownership (Q2) and lifetime of borrows (Q3), the overall average scores were 8.2593 for ownership and 8.1058 for lifetimes. A significant finding came from the question regarding RustLIVE's utility in skill development (Q5), which received the highest average score, 8.5053. The results from all respondents are detailed in Table I.

Also, the standard deviation of 1.4 to 1.7 depicted in Fig. 8 suggests that there was little variation in how respondents rated the tool. Results for each question on the effectiveness of the tool were consistently positive.

A significant finding emerged when comparing the scores provided by participants based on their level of understanding of the Rust memory safety model. Participants who reported a high understanding (level >7)in

TABLE I
EVALUATION RESULTS - ALL RESPONDENTS (GRP1,2,3)

Q1. Understanding Rust	>7	6-7	1-5	all levels
Q2. Ownership	8.78	8.18	7.62	8.26
Q3. Lifetimes	8.52	7.91	7.88	8.10
Q4. Memory Safety	9.03	8.36	7.76	8.43
Q5. Rust Skills	8.85	8.57	7.96	8.51
Total Responses	65	76	49	190

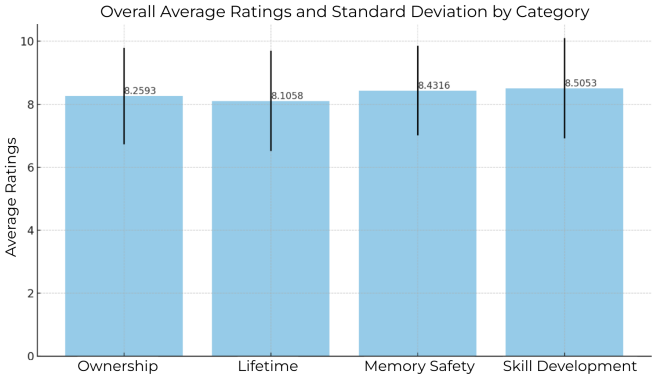


Fig. 8. Overall Ratings with Standard Deviation

Q1 showed a notable correlation with higher ratings for the tool's usefulness compared to those with intermediate understanding (level 6-7) and less experience (level <6). Specifically, individuals with the highest understanding rated the tool as most effective. Further analysis of feedback from the most experienced users revealed positive comments such as, "I think this tool will really help with learning as well as debugging. What a great tool!", "I absolutely think this tool does a great job of visualizing a difficult concept in an intuitive way", and "I believe it will greatly help new devs in the future." Feedback from inexperienced users was less persuasive, with comments such as, "Rust seems to have a steep learning curve and a visual tool is the perfect way to help overcome that steep curve. My understanding is limited but overall the visual aid helped greatly" and "I don't understand rust well, but I think this tool does a great job of visualizing ownership and borrows". Overall, the feedback suggests that individuals with prior Rust coding experience perceive the tool's value more than those without such experience. The results also indicate that these concepts remain challenging even for experienced Rust programmers. Nonetheless, even participants with no prior Rust experience rated the tool's usefulness above 7 on average.

VI. Discussion and Future Work

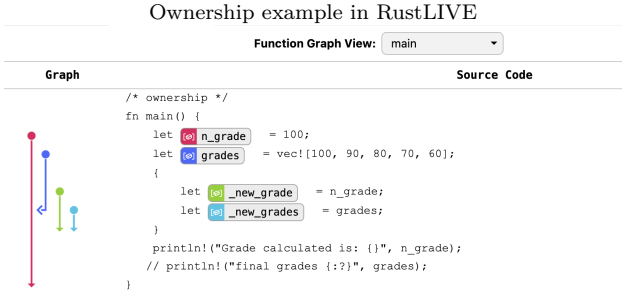
In the Visual Studio Code (VS Code) workspace, programmers can compile their Rust code using any version of the Rust compiler. However, RustLIVE's backend component, the compiler-query, specifically requires rustc 1.75.0-nightly. This nightly version is necessary to gen-

TABLE II
EXAMPLE RUST PROGRAMS FOR ASSESSMENT

Ownership example in VS Code

```
fn main() {
    let n_grade: i32 = 100;
    let grades: Vec<i32> = vec![100, 90, 80, 70, 60];
    {
        let _new_grade: i32 = n_grade;
        let _new_grades: Vec<i32> = grades;
    }
    println!("Grade calculated is: {}", n_grade);
    // println!("Final grades: {:?}", grades);  <-- error
}
```

Ownership example in RustLIVE



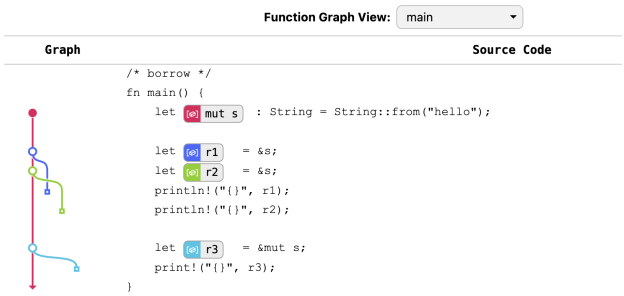
Lifetime example in VS Code

```
fn main() {
    let mut s: String = String::from("hello");

    let r1: &String = &s;
    let r2: &String = &s;
    println!("{}", r1);
    println!("{}", r2);

    let r3: &mut String = &mut s;
    print!("{}", r3);
}
```

Lifetime example in RustLIVE



Memory Safety example in VS Code

```
fn main() {
    let s: String = String::from("Hello");
    let u: String = s;
    let mut t: String = String::from("world");
    t.push(ch: '!');
    let v: &mut String = &mut t;
    v.push(ch: '!');
    let w: &String = &*v;
    println!("{}", w);
}
```

Memory Safety example in RustLIVE

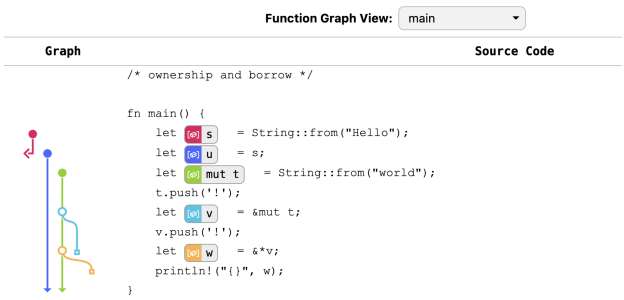


TABLE III
EVALUATION RESULTS - PRIOR RUST KNOWLEDGE (GRP1,2)

Q1. Understanding of Rust	>7	6-7	1-5	all levels
Q2. Ownership	9.40	8.46	7.88	8.46
Q3. Lifetimes	9.40	8.62	8.00	8.58
Q4. Memory Safety	9.60	8.54	8.00	8.58
Q5. Rust Skills	9.40	9.08	8.25	8.88
# of Responses	5	13	8	26

TABLE IV
EVALUATION RESULTS - NO FORMAL RUST EXPERIENCE (GRP3)

Q1. Understanding of Rust	>7	6-7	1-5	all levels
Q2. Ownership	8.73	8.13	7.63	8.23
Q3. Lifetimes	8.44	7.76	7.85	8.03
Q4. Memory Safety	8.98	8.32	7.71	8.41
Q5. Rust Skills	8.80	8.46	7.90	8.45
# of Responses	60	63	41	164

erate the essential data for RustLIVE, yet programmers retain the flexibility to compile with any version, ensuring that RustLIVE remains functional across compiler updates.

In order to maintain simplicity in the graphic timeline, a visual distinction between immutable and mutable owned variables and borrows was not implemented. Future versions could easily incorporate this with a clear design for differentiation. Another idea for future development is to provide tooltips with descriptive information when hovering over points in the graph.

RustLIVE generates a graphic display of one function at a time in the webview panel, defaulting to the `main` function, although other functions can also be depicted. Another suggestion made during the evaluation was to include explanations for compiler errors. However, the data for RustLIVE is generated late in the compilation process so is not generally available if a compilation error occurs.

VII. Conclusion

Research supports a widely held view that many Computer Science students do not have sufficient understanding of memory safety issues or how to prevent them in their programs. Industry leaders also acknowledge that memory safety is a significant concern in modern software. Rust offers a promising solution to these memory safety vulnerabilities, but its adoption is hindered by its steep learning curve. RustLIVE, with its innovative design, aims to clarify Rust's challenging concepts for beginners, particularly ownership and borrowing. Learning to program in Rust not only provides practice in writing memory safe code but also instills safe practices that lead to more robust systems programming in any language. RustLIVE facilitates independent learning and makes integrating Rust into an educational curricula more practical.

References

[1] C. Cimpanu, "Microsoft: 70 percent of all security bugs are memory safety issues," https://www.zdnet.com/article/microsoft-70-percent-of-all-security-bugs-are-memory-safety-issues, 2019.

[2] "Chrome: 70 percent of all security bugs are memory safety issues," https://www.chromium.org/Home/chromium-security/memory-safety, 2020.

[3] J. Lam, E. Fang, M. Almansoori, R. Chatterjee, and A. G. S. Raj, "Identifying gaps in the secure programming knowledge and skills of students," in *Proceedings of the 53rd ACM Technical Symposium on Computer Science Education V. 1*, 2022, pp. 703–709.

[4] L. Szekeres, M. Payer, T. Wei, and D. Song, "Sok: Eternal war in memory," in *2013 IEEE Symposium on Security and Privacy*. IEEE, 2013, pp. 48–62.

[5] "NSA Releases Guidance on How to Protect Against Software Memory Safety Issues," https://www.nsa.gov/Press-Room/News-Highlights/Article/Article/3215760/nsa-releases-guidance-on-how-to-protect-against-software-memory-safety-issues/.

[6] A. D. Keromytis, "Recommendations from the workshop on open-source software security initiative," Georgia Institute of Technology, Tech. Rep., 2022. [Online]. Available: https://cpb-us-w2.wpmucdn.com/sites.gatech.edu/dist/a/2878/files/2022/10/OSSI-Final-Report.pdf

[7] S. Klabnik and C. Nichols, *The Rust programming language (Covers Rust 2018)*. No Starch Press, 2019.

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

[9] D. Gračanin, K. Matković, and M. Eltoweissy, "Software visualization," *Innovations in Systems and Software Engineering*, vol. 1, no. 2, pp. 221–230, 2005.

[10] J. Sorva, "Students' understandings of storing objects," in *Proceedings of the Seventh Baltic Sea Conference on Computing Education Research*, 2007, pp. 127–135.

[11] ——, "The same but different: Students' understandings of primitive and object variables," in *Proceedings of the 8th International Conference on Computing Education Research*, 2008, pp. 5–15.

[12] J. Sorva *et al.*, *Visual program simulation in introductory programming education*. Aalto University, 2012.

[13] J. Sorva, V. Karavirta, and L. Malmi, "A review of generic program visualization systems for introductory programming education," *ACM Transactions on Computing Education (TOCE)*, vol. 13, no. 4, pp. 1–64, 2013.

[14] P. J. Guo, "Online python tutor: Embeddable web-based program visualization for cs education," in *Proceeding of the 44th ACM Technical Symposium on Computer Science Education*, 2013, pp. 579–584.

[15] G. Luo, V. Reddy, M. Almeida, Y. Zhu, K. Du, and C. Omar, "Rustviz: Interactively visualizing ownership and borrowing," *arXiv preprint arXiv:2011.09012*, 2020.

[16] Z. Zhang, B. Qin, Y. Chen, L. Song, and Y. Zhang, "Vrlifetime–an ide tool to avoid concurrency and memory bugs in rust," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020, pp. 2085–2087.

[17] D. Blaser, "Simple explanation of complex lifetime errors in rust," *Bachelor Thesis*, 2019.

[18] D. Dominik, "Visualization of lifetime constraints in rust," *Bachelor Thesis*, 2018.

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

[20] A. Zeng and W. Crichton, "Identifying barriers to adoption for rust through online discourse," *arXiv preprint arXiv:1901.01001*, 2019.

[21] N. Shrestha, C. Botta, T. Barik, and C. Parnin, "Here we go again: Why is it difficult for developers to learn another programming language?" in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 691–701.

[22] "Lifetimes," https://web.mit.edu/rust-lang_v1.25/arch/amd64_ubuntu1404/share/doc/rust/html/book/first-edition/lifetimes.html.

[23] B. Qin, Y. Chen, Z. Yu, L. Song, and Y. Zhang, "Understanding memory and thread safety practices and issues in real-world rust programs," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2020, pp. 763–779.

[24] W. Crichton, "The usability of ownership," *arXiv preprint arXiv:2011.06171*, 2020.

[25] K. Ferdowsi, "The usability of advanced type systems: Rust as a case study," *arXiv preprint arXiv:2301.02308*, 2023.

[26] P. Ruffwind, "Graphical depiction of ownership and borrowing in rust," *Retrieved September*, vol. 13, p. 2020, 2017.

[27] J. Walker, "Rust lifetime visualization ideas," 2019. [Online]. Available: https://blog.adamant-lang.org/2019/rust-lifetime-visualization-ideas/

[28] P. D. Faria, "Borrow visualizer for the rust language service," *Retrieved September*, vol. 13, p. 2019, 2019.

[29] T. R. S. Team, "2023 rust annual survey results," https://blog.rust-lang.org/2024/02/19/2023-Rust-Annual-Survey-2023-results.html, Feb 2024, accessed: 2024-05-13.

[30] JetBrains, "Developer ecosystem survey 2023," https://www.jetbrains.com/lp/devecosystem-2023/, 2023, accessed: 2024-05-13.

[31] "Rust compiler development guide," https://rustc-dev-guide.rust-lang.org/overview.html#mir-lowering.

[32] N. Matsakis, "An alias-based formulation of the borrow checker," https://smallcultfollowing.com/babysteps/blog/2018/04/27/an-alias-based-formulation-of-the-borrow-checker/, Apr 2018, accessed: 2024-05-13.