RustViz: Interactively Visualizing Ownership and Borrowing

Marcelo Almeida, Grant Cole, Ke Du, Gongming Luo, Shulin Pan, Yu Pan,
Kai Qiu, Vishnu Reddy, Haochen Zhang, Yingying Zhu, Cyrus Omar
University of Michigan
Ann Arbor, MI, USA
{mgba,gtcole,madoka,luogm,shulinp,panyu,qiuk,reddyvis,haochenz,zyy,comar}@umich.edu

Abstract—**Rust is an industrial systems programming language unique in achieving memory safety without the need for a garbage collector. Instead, Rust relies on a unique and sometimes subtle resource ownership and borrowing system. This system can make learning Rust a challenge, even for experienced programmers. Motivated by these challenges, we introduce RustViz, a tool that allows an instructor to generate custom interactive timelines depicting ownership and borrowing events alongside Rust code examples embedded within learning material. These visualizations makes visible the static events, and subsequent state changes, that a Rust programmer must otherwise track entirely mentally. We have used RustViz to build a week-long Rust unit in a large undergraduate programming languages course. We demonstrate that this learning material, and the RustViz visualizations in particular, were valuable to students and led to the development of an accurate mental model of the essentials of ownership and borrowing in Rust.**

I. Introduction

Today, critical systems software (e.g. operating system components, web browser renderers, drivers, and numerical libraries) is typically written in languages like C and C++ that are not memory safe. Programs can, for example, read from memory that has already been freed or access an out-of-bounds location in an array. This can cause program crashes, memory corruption, and open up security vulnerabilities. The costs of bugs like these are collectively enormous: Microsoft reported that 70% or more of all security vulnerabilities are caused by memory safety issues [1]. In addition to safety issues, C/C++ programs often exhibit memory leaks, where resources are not deallocated though they are no longer referenced.

The most direct solution to these pernicious problems is to write systems software in languages that enforce the use of a memory safe automatic memory management system. Indeed, nearly every mainstream language today except for C and C++ is memory safe. However, memory safety typically comes at a cost: automatic memory management typically requires a run-time garbage collector. While garbage collection algorithms have advanced significantly over the past several decades, the associated run-time cost is still perceived to be prohibitive for performance-critical systems software [2].

Rust sits in a unique niche: it is a systems programming language that achieves memory safety *without* relying on run-time garbage collection [3, 4]. Instead, Rust's automatic memory management is achieved by an entirely static (compile-time) analysis, so there is no run-time cost. In particular, Rust's *borrow checker* statically associates a unique *owner* with each allocated resource. The resource is automatically deallocated ("dropped", in Rust parlance) when its owner goes out of scope. The key complication is that ownership of a resource is moved when passing data into and returning from functions. To increase flexibility, the owner of a resource can also enable borrowed access to the resource, without giving up ownership, via references (internally, pointers). The borrow checker enforces a set of rules to ensure that such borrows do not outlive the resource's owner, thereby ensuring memory safety. Reference lifetimes are non-lexical, determined by a last-usage analysis within their lexical scope.

In addition to memory safety, Rust enforces a form of thread safety (avoidance of data races, another notorious source of bugs in systems software), aliasing control, and mutation control by distinguish two kinds of references. Mutable (a.k.a. unique) references allow for mutation, but within their lifetime the borrow checker ensures that they are unique in their access to the resource. Elsewhere, any number of immutable (a.k.a. shared) references can be created.

Despite these unique and subtle restrictions, Rust has been gaining traction explosively. For five consecutive years, Rust was the "most loved" programming language in Stack Overflow's Developer Survey [5]. In industry, Rust has notable users such as Microsoft [6], Amazon [7], Mozilla [8], Cloudflare [9], Dropbox [10], and Discord [11].

However, the subtlety inherent in Rust's ownership and borrowing rules also make Rust difficult to learn and Rust code difficult to reason about, even for experienced programmers [12, 13]. A systematic survey of posts on various online forums found that people learning Rust frequently complained that the borrow checker was inaccessible [14]. In interviews, participants learning Rust reported that the borrow checker was an "alien concept" and "the biggest struggle" in learning Rust [15]. Searches on Rust forums show that "fighting the borrow checker" has become a common refrain, particularly for novices. Interviews with industry professionals also found that difficulties with learning Rust presented barriers to adoption [2]. A participant in a study by Coblenz et al. [16] writes:

978-1-6654-4214-5/22/$31.00 ©2022 IEEE

“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.

Motivated by the difficulties that Rust learners face, we sought to develop learning material targeted at students with at least 2 semesters of programming experience (CS1 and CS2, taught using C++ at our institution), but no exposure to Rust.

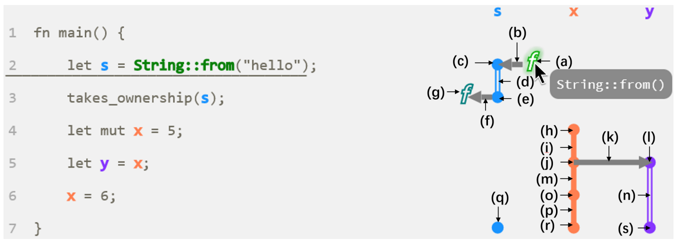
Based on the studies above, we identified the key challenge: the user must learn to mentally simulate the logic of the borrow checker, which is known to be challenging [17–19]. The source code itself does not directly express key events and quantities, e.g. ownership movement events, reference lifetimes, or uniqueness-related constraints. 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 in the form an interactive timeline for each variable, displayed aligned with Rust source code examples appearing in written learning material.

There are two kinds of RustViz users: *instructors* use RustViz to generate visualizations for integration into learning material and *learners* interact with the visualizations while going through learning material. Sec. II introduces RustViz from the learner’s perspective (and serves as a brief introduction to Rust itself). Sec. III then describes how instructors can prepare such visualizations, customized to focus specifically on the program elements of interest, by annotating Rust source code using a simple domain-specific language (DSL).

We have integrated RustViz into a two-lecture Rust unit in an upper-level undergraduate course on Programming Languages taken by 313 students over four semesters. Sec. IV describes the learning goals and reports the results of quantitative assessments, qualitative surveys, and thinkaloud lab sessions involving the participants. This data suggest that the students, who all have prior experience with C++ and limited or no prior exposure to Rust, are able to achieve the intended learning goals, i.e. they are able to form an accurate mental model of the essential components of Rust’s distinctive ownership and borrowing system over the course of only one week of instruction and one homework assignment. The RustViz visualizations were extensively used by, and considered valuable by, these students. Students were largely enthusiastic about the language after engaging with this learning material.

In addition to the required assignment, students from one semester were offered extra credit to generate a RustViz visualization, taking on the role of a hypothetical instructor. Students who submitted this exercise were able to generate the correct visualization within two hours. Some students reported that this helped improve their understanding of Rust itself, suggesting a pedagogical role for visualization generation.

Taken together and more broadly, this data suggests that in learning advanced type systems and static analyses, visualization can serve a valuable role as part of a broader pedagogical context. We conclude after a discussion of future and related work in Secs. V-VII.



```
1  fn main() {
2      let s = String::from("hello");
3      takes_ownership(s);
4      let mut x = 5;
5      let y = x;
6      x = 6;
7  }
```

Fig. 1. Moves, Copies, and Drops (sublabels for exposition)

II. RustViz by Example

RustViz visualizations consist of a piece of Rust source code, typically integrated into a narrative tutorial or book chapter (generated by the popular `mdbook` tool, see Sec. III), together with an adjacent *timeline* of memory-related *events* for each variable in the source code that the instructor chooses to visualize, in an instructor-specified order. This timeline is displayed aligned with the corresponding source code. Color is used as secondary notation to better visually group source code elements with the corresponding column of the timeline. When learners hover their cursor over the various visual elements of the timeline, Hover Messages are displayed which describe the memory events in more detail. To avoid visual clutter, we describe the key Hover Messages in the text by referring to sublabels in the figures (which do not appear to users).

A. First Example: Moves, Copies and Drops

Fig. 1 shows a simple example that demonstrates *moves*, *copies*, and *drops*, all fundamental Rust concepts.

1) Moves: On Line **2**, a heap-allocated `String` resource is created and bound to `s`. In Rust, each resource has a unique *owner*, and when ownership changes, it is called a *move*. In this case, `String::from` heap-allocates and returns a `String`, which moves ownership of that resource to `s` as indicated by the Arrow at **(b)** pointing from **(a)** to the Dot at **(c)**. The hollow Line Segment at **(d)** indicates that `s` cannot be reassigned nor can it be used to mutate the resource because `let` rather than `let mut` was used to define `s`. As the learner hovers their mouse over these visual elements, the corresponding line is underlined and the following Hover Messages are displayed:

- **(a)** `String::from()` (the hover message and highlighted function name and line are shown in Fig. 1 as an example)
- **(b)** *Move from* `String::from()` *to* `s`
- **(c)** `s` *acquires ownership of a resource*
- **(d)** `s` *is the owner of the resource. The binding cannot be reassigned.*

On Line **3**, `takes_ownership` (not shown) is called with `s` as a parameter, so the `String` resource gets moved from `s` into the function. After this move, `s` is no longer the owner of the resource, so the resource is no longer accessible through `s`. This move is shown in RustViz with the Arrow at **(f)** from **(e)** to **(g)**. Since `s` is no longer valid for use after the move event, there is no longer a visible Line Segment in `s`’s timeline after the Dot. The Hover Messages here are:

(e) *s's resource is moved*
(f) *Move from* s *to* takes_ownership()
(g) takes_ownership() (and the corresponding function in the code is highlighted)

2) Copies: On Line **4**, the (immutable) integer value 5 is bound to x. We use let mut rather than let, so x can be reassigned. as indicated by the solid Line Segment at **(i)**. The Hover Messages are:

(h) x *acquires ownership of a resource*
(i) x *is the owner of the resource. The binding can be reassigned.*

On Line **5**, y is initialized with x. In Rust, types with stack-only data, like integers, generally have an annotation called the Copy trait. Resources of these types get *copied* rather than moved. The copy is shown in RustViz with the Arrow at **(k)** pointing from **(j)** to **(l)**. Since x is still valid for use, the solid Line Segment continues in x's timeline at **(m)**. The Line Segment at **(n)** is hollow because we used let rather than let mut for y. The Hover Messages here are:

(j) *x's resource is copied*
(k) *Copy from* x *to* y
(l) y *is initialized by copy from* x
(m) x *is the owner of the resource. The binding can be reassigned.*
(n) y *is the owner of the resource. The binding cannot be reassigned.*

On Line **6**, we mutate x, as indicated by the Dot at **(o)** continuing to solid Line Segment **(p)**:

(o) x *acquires ownership of a resource*
(p) x *is the owner of the resource. The binding can be reassigned.*

3) Drops: The variables s, x, and y go out of scope at the end of the function on Line **7**, as indicated by Dots **(q)**-**(s)**. Since x and y were owners, their resources are *dropped*. However, since s's resource was moved earlier, no drop occurs, as indicated in the corresponding Hover Messages:

(q) s *goes out of scope. No resource is dropped.*
(r) x *goes out of scope. Its resource is dropped.*
(s) y *goes out of scope. Its resource is dropped.*

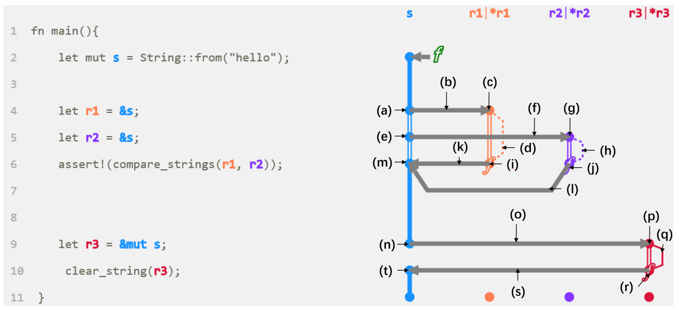
By following the arrows coming out of move events, it is possible to see exactly when each resource is automatically dropped by Rust.

B. Second Example: Shared and Unique Borrows

Our second example, in Fig. 2, demonstrates *borrowing*, i.e. working with references to resources.

1) Immutable Borrows: On Line **2**, a mutable String resource is created as described in Sec. II-A.

On Lines **4** and **5**, *immutable references* (a.k.a. *shared* references) to s are created and assigned to r1 and r2, respectively. In Rust, immutable references are created with the & operator and the creation of an immutable reference is called an *immutable borrow*. Resources cannot be mutated through immutable references, so it is thread safe for Rust's borrow checker to allow multiple immutable borrows of a



```
1  fn main(){
2      let mut s = String::from("hello");
3
4      let r1 = &s;
5      let r2 = &s;
6      assert!(compare_strings(r1, r2));
7
8
9      let r3 = &mut s;
10      clear_string(r3);
11  }
```

Fig. 2. Borrows

resource to be live. The timelines for variables of reference type are split into two parts. The first part is the Line Segment that represents the variable itself, which is displayed in the same way as variables of other types. The second part is the curved line to the right of the Line Segment, here **(d)** and **(h)**, which represents how the borrow allows access to the resource, i.e. how ⋆r1 and ⋆r2 work as indicated in the header. The Hover Messages are:

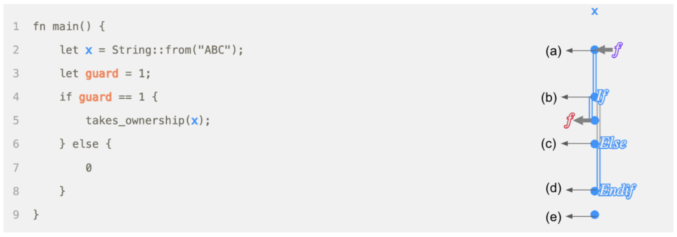
(a) *s's resource is immutably borrowed*
(b) *Immutable borrow from* s *to* r1
(c) r1 *immutably borrows a resource*
(d) *Cannot mutate* ⋆r1
(e) *s's resource is immutably borrowed*
(f) *Immutable borrow from* s *to* r2
(g) r2 *immutably borrows a resource*
(h) *Cannot mutate* ⋆r2

On Line **6**, r1 and r2 are passed to compare_strings, which is shown in RustViz by the *f* symbol on **(i)** and **(j)**. compare_strings can read the resource through these references, but cannot mutate it. Since r1 and r2's borrows are no longer used after the function returns, the borrowed resource is returned, despite r1 and r2's lexical scope not ending. This is due to a feature in Rust called *non-lexical lifetimes*, which allows more programs to pass the borrow checker by disregarding borrows which are no longer live because they are never used again. The return of these borrows are represented by the Arrows at **(k)** and **(l)**. Some Hover Messages here are:

(i) compare_strings() *reads from* r1
(j) compare_strings() *reads from* r2
(k) *Return immutably borrowed resource from* r1 *to* s
(l) *Return immutably borrowed resource from* r2 *to* s
(m) *s's resource is no longer immutably borrowed*

Variables cannot be mutated if there are live borrows. This is indicated by the change from a solid to a hollow line segment at the first borrow event, **(a)**. Once all of the borrows have been returned, the line segment is solid again.

2) Mutable Borrows: On Line **9**, a *mutable reference* (a.k.a. *unique reference*) to s's resource is created and assigned to r3. In Rust, mutable references are created with the &mut operator rather than the & operator, and the creation of a mutable reference to a resource is called a *mutable borrow*. Resources



```
1  fn main() {
2      let x = String::from("ABC");
3      let guard = 1;
4      if guard == 1 {
5          takes_ownership(x);
6      } else {
7          0
8      }
9  }
```

Fig. 3. Conditionals

can be mutated through mutable references. To ensure thread safety and help control aliasing, if a mutable borrow of a resource is live, no other borrow of that resource, mutable or immutable, may be live, nor may the resource be accessed by the owner. Thanks to non-lexical lifetimes, `r3`'s borrow does not conflict with `r1` or `r2` because `r1` and `r2`'s borrows are not live. We show that `r3`'s borrow is mutable in RustViz with a Solid Line at **(q)**. Since `r3` mutably borrows s's resource, we cannot access the resource through `s`, so there is no Line Segment between Lines **9** and **10**. The Hover Messages here are:

- **(n)** *`s`'s resource is mutably borrowed*
- **(o)** *mutable borrow from `s` to `r3`*
- **(p)** *`r3` mutably borrows a resource*
- **(q)** *Can mutate the resource *r3*

On Line **10**, `r3` is passed to `clear_string`, and `clear_string` is able to mutate the `String` through that reference. After the function returns, the borrowed resource is returned to `s`. With the `String` resource no longer being mutably borrowed, `s`'s Line Segment in the timeline resumes. Some Hover Messages here are:

- **(r)** *`clear_string()` reads from `r3`*
- **(s)** *Return mutably borrowed resource from `r3` to `s`*
- **(t)** *`s`'s resource is no longer mutably borrowed*

C. Third Example: Conditionals

Our third example, in Fig. 3, demonstrates how ownership changes in *conditionals* are visualized. The key idea is that the ownership state at the branchpoint, **(b)**, is the initial state in each branch. However, the visualization challenge is that the `else` branch is far away from the branchpoint. To communicate this, we mark branching in the visualization by splitting the timeline. In the first branch, the left split is active and visualizes the move on Line 5, at **(c)**. In the `else` branch, the right split, which was left grayed out but visible to indicate continuity from the branchpoint, is active. In this branch, no move occurs. At the end of the conditional, the ownership state has been moved if either branch ended with the resource being moved away. Again, we use the Hover Messages to indicate the semantics:

- **(a)** *`x` acquires ownership of a resource*
- **(b)** *Conditional is entered. `x` owns a resource at the start of both branches.*
- **(c)** *`x` owns a resource at the start of the `else` branch*
- **(d)** *One branch caused a change in ownership, so `x` no longer owns a resource. Its resource will be dropped if it was not moved.*
- **(e)** *`x` goes out of scope. No resource is dropped.*

III. Generating RustViz Visualizations

We now turn to the instructor's perspective. Instructors generate visualizations using a domain-specific language (DSL) written in comments within the Rust code being visualized, which are read and stripped by the RustViz tool. Consider as a running example the simple visualization in Fig. 4a. The corresponding visualization specification is in Fig. III. The visualization specification consists of a:

1) Column Specification, which specifies which variables and related functions are to be visualized; and
2) Event Specifications, which specify ownership and borrowing events via structured comments after each corresponding line

A. Column Specification

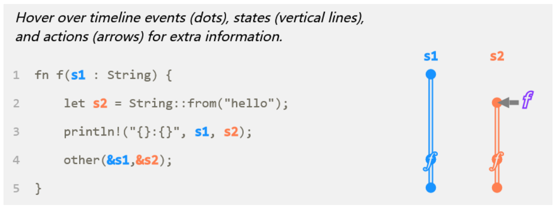
The column specification at the top of Fig. III specifies names of all the variables that are used in the visualization together with their role: `Owners`, `StaticRefs` or `MutRefs`, or `Functions`. The first three of these roles appear as columns in the order specified. `Functions` can be used for the function icons that appear between columns.

A column specification including all variables and functions mentioned in the code can be automatically generated from a Rust file as a starting point. However, typically there will be variables or functions that are not of pedagogical interest (e.g. the `println!` macro function, or `guard` in Fig. 3), so after automatic generation, the instructor is free to manually remove or re-order the columns as necessary.

B. Event Specification

Once the columns are specified, the instructor can write an event specification via structured comments at the end of each line where one or more events occur. Each structured comment consists of comma-separated event descriptions between `!{` and `}`. Each event description consists of the event name and its parameters.

On Line 7, the event `InitResourceParam(s1)` specifies that `s1` acquires ownership of a resource from the caller because it is a function parameter. On Line 8, the event `Move(String::from()->s2)` specifies that there is a move from `String::from()` to `s`. The instructor decided not to visualize events related to `println!()` on Line 9. On Lines 10-11, the events `PassByStaticReference(s1->other())`, `PassByStaticReference(s2->other())` specify that both `s1` and `s2` are statically passed to function `other()` as references. On Line 12, the events `GoOutOfScope(s1)`, `GoOutOfScope(s2)` specify that `s1` and `s2` are going out of scope. The system automatically determines whether a drop will occur based on the prior move events.


Hover over timeline events (dots), states (vertical lines), and actions (arrows) for extra information.

```
fn f(s1 : String) {
    let s2 = String::from("hello");
    println!("{}:{}", s1, s2);
    other(&s1,&s2);
}
```

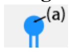
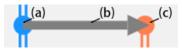
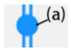
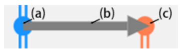
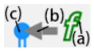
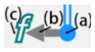
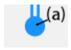
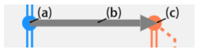
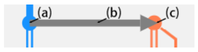
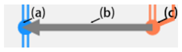
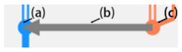
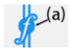
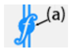
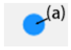
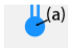
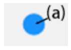
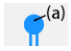
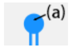
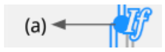
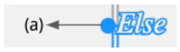
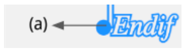
(a) Example Visualization Output

```
/*--- 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) } */
\end{lstlisting}
```

(b) Corresponding Visualization Specification

Fig. 4. Generating RustViz 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.
`InitRefParam(x)`		(a) `x` is initialized as the function argument.
`InitResourceParam(x)`		(a) `x` acquires ownership of a resource.
`StartIf()`		(a) Enters an if block.
`StartElse()`		(a) Enters an else block.
`EndIf()`		(a) Finishes an entire conditional block.

The events that Rustviz supports are included in Table I, together with their hover messages and the visual depiction that is generated. Lines in this table are shown as hollow unless it is obligatory for the line to be solid. Events related to closures and structs are also supported, but were not used in our evaluation or presented in the previous section, so we omit them from the table.

The system performs lightweight error checking for events. Namely, DSL syntax errors are reported, as are uses of names that were not declared in the Column Specification. However, no semantic analysis of the Rust code itself is performed.

C. Visualization Generation

The instructor supplies this annotated source file, i.e. the visualization specification, to the RustViz DSL processor to generate SVG files containing the original source code (with DSL comments stripped) and the corresponding timeline visualization. These SVG files can be incorporated into narratives generated using a version of *mdbook*, a popular documentation system in the Rust community (used, for example, by the official Rust Book), equipped with an additional helper script that handles hover interactions. The resulting book can be viewed in a web browser. The visualizations could also in principle be integrated into any other webpage.

IV. Evaluation

We evaluated the effectiveness of RustViz within a broader pedagogical context: a short Rust unit, consisting of two 80 minute lectures, an optional 60 minute discussion section, a 30-minute written tutorial, and a homework assignment, which included a survey. This unit has been integrated into the second half of an undergraduate elective Programming Languages course. RustViz is integrated into the tutorial material directly. In addition, the lecture material includes hand-drawn whiteboard diagrams inspired by RustViz's visual language. Our quantitative focus was on evaluating the learning unit as a whole, rather than on isolating the impact of specific visualization design choices. We evaluated the visualization design primarily via qualitative means, discussed below.

A. Learning Goals

This unit was near the end of the semester, so students were assumed to have basic understanding of basic programming language concepts, e.g. scope and memory. We sought to achieve the following learning goals:

1) Understand concepts related to ownership in Rust
    a) Understand that each resource has a unique identifier called its owner
    b) Understand the different ways that ownership can be moved
    c) Be able to determine when resources are automatically dropped (when the owner goes out of scope)
    d) Understand the difference between moves and copies
2) Understand Rust's borrowing rules
    a) Understand the difference between mutable and immutable borrows
    b) Understand that borrows cannot outlive the owner
    c) Understand non-lexical lifetimes

Notably, we avoid certain advanced Rust concepts such as lifetime annotations, compound data types (e.g., structs and array slices), reborrowing, and `unsafe` code.

B. Study Format

The participants in our study are students of an elective Programming Language course for computer science majors. This course was taught four times, over four consecutive semesters (the first two of which were taught remotely due to the COVID-19 pandemic), consisting of 62, 65, 74, and 112 students, respectively, totaling 303 students. The lecture and tutorial materials, and certain aspects of the visualization (e.g. spacing between the code and the visualization, the line spacing, the ease of hovering over lines and other small elements, the addition of the line underlining when hovering) were modified somewhat between semesters based on student feedback, though the overall structure of the material did not change, so we analyzed the results from each semester separately. However, the results below are pooled because we did not observe substantial qualitative differences in response distributions or performance. The study was conducted with Institutional Review Board (IRB) approval. All participants are adults who are proficient in C++ (which is taught in the two-course pre-requisite sequence) but report limited or no prior exposure to Rust: 31.9% of the students had not heard of Rust at all, an additional 52.4% had heard of Rust but not learned any details of its semantics, while less than 5% had more than trivial experience with Rust.

We additionally selected four paid student volunteers for a thinkaloud study to provide additional qualitative context. These participants were recorded as they engaged with the tutorial and assignment questions for one hour and asked to "think aloud" as they engaged with the tutorial text and visualizations, describing their thought processes, including times when they were confused or unsure. Participants were also asked to describe their problem solving process when answering corresponding assignment questions.

C. Learning Material

Prior to engaging in the assessment portion of the Rust unit, participants were able to virtually attend or asynchronously watch two 80-minute lectures introducing Rust's ownership and borrowing system. These lectures were optional, and between 20-30% of students did not attend or watch the lecture videos before starting on the assignment.

In addition to the optional lectures, the students were required to read an interactive tutorial, developed using *mdbook* and RustViz as described in Sec. III. This tutorial was designed to take about 30 minutes and covered the same concepts as lecture, in more specific detail. The tutorial text and code examples were designed to convey all of the necessary concepts in isolation, even if the student did not refer to or understand the visualizations. Students generally spent between 20-45 minutes on the tutorial as assessed by both self-report and telemetry, and almost no students skipped the tutorial.

In the survey after the assessment, students were asked to reflect on the helpfulness of the tutorial text and the interactive visualizations separately. Rating on a 5-point Likert scale, The results show that nearly all students found both the text and the interactive visualizations helpful or very helpful. In particular, across four semesters, 47% reported that the interactive visualizations in the tutorial was very helpful in terms of improving their understanding of ownership and borrowing in rust, and 46% reported it to be helpful.

When asked to assess the level of interaction that students had with the RustViz visualizations in particular, about 3 out of 4 students reported either interacting with each code visualization or regularly inspecting the visualizations throughout the tutorial. In our thinkaloud sessions, the participants chose not to inspect all segments of the visualizations on the first pass. If the reading was confusing, the participants would generally spend more time on the corresponding visualization, or would revert back to a visualization from earlier in the tutorial. All thinkaloud participants provided feedback suggesting that they found the visualizations helpful in these particular situations. Free response feedback also indicated that students referred back to visualizations multiple times:

> *"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."*

Much of the feedback focused on the difficulty of pointing at a particular part of the visualization that was of interest, due to crowding. We made small improvements between semesters but a more comprehensive solution remains desirable:

> *"The arrow heads seemed to overlap a bit as well, so it was harder to see some of the directions of actions."*

> *"The hovering required precision sometimes, like in the Borrowing section where there were multiple things that can be hovered clustered together. Maybe bigger or more spaced out targets would help with this."*

In addition, some students noted difficulty with understanding the meaning of certain aspects of the visual notation, which could perhaps be addressed using a key or making this more explicit in the tutorial text when the corresponding concept is introduced:

> *"A key identifying what different types of lines and symbols meant would have been helpful. I still don't know what the double line vs the single line means in the diagrams."*

> *"I wish there was an easier way to tell if the lines represented ownership, immutable borrowing, or mutable borrowing by just looking at the visuals without hovering my mouse over them. Maybe a key of some sort for what each line represents? I like the hover feature but maybe there is a way to provide more detail at a glance."*

D. Exercises

After engaging with the learning material, students were asked to complete a series of graded multiple choice exercises. Students were allowed to refer back to the learning material freely during these exercises. Students were not informed of whether their answer was correct until after the deadline.

The exercises were designed to assess the learning goals described above, while avoiding questions where the answer could be produced simply by entering the code into a Rust REPL. For example, to assess students' ability to reason about ownership changes and drops, we asked questions like:
Consider the following Rust program.

```rust
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.");
}
```

The student is then provided with the text that is printed to standard out, followed by the following multiple-choice question. "*The string resource that* `a` *initially owns is dropped between which two adjacent lines of output above?*". In the example above, the answer is just after `Goodbye, world!`. This cannot be determined simply by entering the code into Rust, because drops do not elicit any visible side effect.

As a free response follow up, we ask the student to "*Explain why by describing the events related to movement and borrowing that occur in the code above.*"

It took students a median of 5.43 minutes to provide an answer to each exercise. Students performed well: the average score for the exercises was 5.46 out of 6, with a standard deviation of 0.662. The was one question that students performed most poorly on (about 30% of students answered incorrectly) required an understanding of non-lexical lifetimes, suggesting a need to improve coverage of this topic.

We assessed the free response question qualitatively, and found that students were generally correct in the sequence of moves and borrows involved in each question, though the precision in the language used varied (we did not provide any examples). A slight preference for phrases used in the Hover messages was apparent, but we did not analyze this quantitatively.

Students did generally choose to refer back to the tutorial while answering the exercises. About 30% of students also reported accessing other resources (e.g. an online IDE, the official documentation, or other tutorials or books). Though most students had no experience using Rust coming into the unit, the survey after the assessment indicated that over 85% of the students were interested or very interested in using the Rust language. This enthusiasm was also evident in survey results at the end of the semester (and in informal interactions).

E. Extra Credit: Creating a Visualization

During the second semester, we also offered an extra credit assignment (several weeks after the above assignment, after one intervening assignment which also used Rust to explore parallelism and concurrency) which allowed students to try generating the visualization for an example program using the RustViz DSL. We assessed the correctness of the generated visualization and included several additional qualitative survey questions. The exercise material is included in the supplement.

Among the students completing the previous assignment, 28 of them completed the extra credit assignment. It is not clear whether there was a bias in favor or against students who were doing well. It took 103.1 minutes on average for a student to complete this assignment, including reading the tutorial, setting up Rust locally, setting up RustViz, annotating the sample source code using our DSL to generate the timelines, and debugging. Over 60% of the students agree or strongly agree that the set up process is simple and easy, while 90% agree or strongly agree that the annotation format is simple and intuitive. Additionally, more than 60% of the students reported that trying to generate visualization for an example using RustViz helps them better understand move, copy and borrow events in Rust programs. This suggests that it may be helpful to use RustViz as a tool for students to generate these examples themselves in order to develop a more systematic understanding of Rust.

F. Threats to Validity

All four semesters of the course described above were taught during the COVID-19 pandemic, with two semesters being taught entirely remotely. The student population in the remote semesters was unlikely to be statistically identical to the population in the first semester, having gone through many months of isolation in a remote setting. Neither is either group necessarily representative of students in a more conventional classroom setting.

While the student population had a background typical of the systems programming domain, with their experience being primarily or exclusively C++ programming, it may be difficult to generalize to, for example, programmers who have been in industry for many years, or to programmers who are not familiar with manual memory management at all.

While we attempted to gather qualitative feedback about each component of our learning materials separately, it is difficult to isolate the impact of any one aspect of our design, and indeed RustViz was designed to operate as one component in a set of educational materials sharing learning goals. Our survey and thinkaloud observations suggested that participants were making active use of the visualizations as part of an information foraging study strategy, and that merely attending lecture had not already prepared them to understand the material in the tutorial. Our deployment in a classroom setting did not allow us to form a control group of students who, for example, read a version of the tutorial without the RustViz visualization. This would be an interesting to study in a laboratory setting in the future.

Our quantitative assessment is not a validated instrument. We hope that more research into assessing understanding of advanced language concepts can be developed to make it possible to compare alternative Rust learning materials.

V. Limitations and Future Work

The presented design does not visualize code that fails the borrow checker, although such code is often helpful in an instructional narrative, including in the tutorial we designed. Rust's error messages already include ASCII diagrams explaining in detail why each error occurs, and these error messages are highly valued by the Rust community.

We also neglected advanced Rust features, which may be of interest in a course that covers Rust more in depth. We created a primitive version of visualization of the conditional branches, described in Sec. II, but we did not include a detailed account of conditionals in the tutorial or assessment. Similarly, while we have designed visualizations for closures (which can take ownership of resources) and structs, these were not evaluated in the context of the one-week unit.

Currently, RustViz generates visualizations from manually-specified memory events rather than automatically from the source code directly. Only the column specification can be automatically generated. We might explore the possibility of automatic generation of memory events from source code. However, this might make it difficult for instructors to specifically control the visualization, e.g. by hiding certain subtleties or well-understood aspects of a code example to focus on the new ideas. A hybrid approach, similar to our Column Specification system, may be helpful. However, we did not find that the work involved in writing the tutorial visualizations was a significant barrier as instructors of the course, and the control offered was useful. Fully automatic visualization generation could, however, be helpful for students or practitioners interested in explanations of code that they have written. Handling the scaling challenges inherent in our visual design (where arrows can overlap and timelines are adjacent) is left as future work.

VI. Related Work

RustViz is a program visualization system. Program visualization, in the pedagogical setting and more broadly, has a long history that has been surveyed extensively [20–22]. RustViz is distinctive in that it focuses on visualizing the *static semantics* of Rust, rather than run-time behavior (where tools like Python Tutor [23] and other algorithm visualization systems have been influential [21]). Visualization systems focusing on static aspects of programs have focused on visualizing syntactic structure, data and control flow, or architectural relationships [24]. A related line of work on type debuggers has focused on helping explain static type errors [25, 26]. RustViz is similar to TypeTool [27] in its focus on explaining correct but subtle code, but differs in that it focuses on borrow checking rather than type checking.

One line of research focused on visualizing the structure of object graphs using a structure called an "ownership tree" [28–30]. However, ownership here relates to encapsulation in object-oriented programming [31], rather than the static resource ownership discipline used as the basis for Rust's memory management system. The ideas here may be relevant to visualizing ownership relationships involving structs in Rust, which we did not consider in detail in this paper.

We find that there is little peer-reviewed research specifically on visualizations to assist in reasoning about Rust's unique ownership and borrowing rules despite interest in the Rust community for such tools. Several blog posts propose various visualizations informally. One showcases a vertical timeline of ownership and borrowing events [32]—this design is visually similar to RustViz, except without interactivity. A different mock-up in another blog post shows a possible approach for visualizing Rust code in an editor rather than for documentation [33], albeit emphasizing lifetime extents rather than the full set of events discussed in this paper. On the Rust Internals Forum [34], a member (username Nashenas88) started a thread to discuss ideas for visualizing ownership and borrowing within an editor. The thread contains various ideas for such a visualization from Faria and others. Notably, the member was able to create a working prototype of lifetime visualization in Atom that is generated directly from Rust source code. In all cases, these mockups were not substantially developed or evaluated.

For their bachelor thesis [35], Dominik developed an algorithm that identifies lifetime constraints and the code that generates the constraints from information extracted from the Rust compiler, including the Polonius borrow checker. Dominik also shows a visualization of the lifetime constraints as a directed graph. Blaser [36] builds on this work for their bachelor thesis by developing a tool that can explain lifetime errors in code given the lifetime constraints. Blaser also created an extension for Visual Studio Code that can show a graph-based visualization of the lifetime constraints that is simpler than Dominik's. As an alternative to the graph-based visualization, Blaser's Visual Studio Code extension can show a text-based explanation of lifetime errors in Rust code. The focus of Blaser's tool is to help programmers reason about lifetime-related errors, while our focus is on generating visualizations in documentation for use in a teaching setting.

Flowistry is a system for visualizing information flows in Rust code, aimed at helping users hide code unrelated to their current task [37]. This slicing approach might be helpful in improving visualizations involving conditionals.

In the direction of visualizations of other "esoteric" semantics, there are some visualization tools that support learning specialized programming languages other than Rust. SQLVis, which is a graph-based visualization tool that displays the implicit relationship between database tables in user-provided SQL queries, proves to be helpful for novice SQL programmers [38]. CUPV help more advanced learners who study compilers to understand operations of generated parsers [39].

VII. Conclusion

In this paper, we introduced RustViz, a tool to generate visualizations of ownership and borrowing in Rust programs from manually-provided memory events. These visualizations are designed to be displayed alongside example code in documentation to help Rust learners develop a basic understanding of Rust's ownership and borrowing rules. We showcased our visual design by walking through some examples, showed how RustViz is used to generate the visualizations, described learning goals that RustViz is being designed to help learners accomplish, and described and evaluated the design of a short Rust unit that has been deployed at scale in the classroom. The results are quite promising – it seems likely that with the right learning material, programmers familiar with C/C++, and perhaps other languages, can learn key ideas unique to Rust in a short period of time, perhaps just an hour or two. This bodes well for the future of memory and thread safety in systems programming applications—perhaps human-centered techniques in conjunction with modern language design can help turn the tide in the battle against memory-related security vulnerabilities, program crashes, and program complexity.

Acknowledgements

The authors would like to thank Ethan Brooks for developing a portion of the Rust coding assignment infrastructure, the course staff of EECS 490 at the University of Michigan (Fall 2020 through Winter 2022) for collaborating with the RustViz team to integrate RustViz into the course infrastructure, and the students of EECS 490 for their participation in the study, their valuable feedback, and their top-notch crab drawings on the final exam.

References

[1] G. Thomas. (2019) A proactive approach to more secure code. [Online]. Available: https://msrc-blog.microsoft.com/2019/07/16/a-proactive-approach-to-more-secure-code

[2] K. R. Fulton, A. Chan, D. Votipka, M. Hicks, and M. L. Mazurek, "Benefits and drawbacks of adopting a secure programming language: Rust as a case study," in *Seventeenth Symposium on Usable Privacy and Security, SOUPS 2021, August 8-10, 2021*, S. Chiasson, Ed. USENIX Association, 2021, pp. 597–616. [Online]. Available: https://www.usenix.org/conference/soups2021/presentation/fulton

[3] N. D. Matsakis and F. S. Klock, "The Rust language," *Ada Lett.*, vol. 34, no. 3, p. 103–104, oct 2014. [Online]. Available: https://doi.org/10.1145/2692956.2663188

[4] R. Jung, J. Jourdan, R. Krebbers, and D. Dreyer, "Safe systems programming in Rust," *Commun. ACM*, vol. 64, no. 4, pp. 144–152, 2021. [Online]. Available: https://doi.org/10.1145/3418295

[5] S. Overflow. (2020) Stack overflow developer survey 2020. [Online]. Available: https://insights.stackoverflow.com/survey/2020

[6] R. Vengalil. (2019) Building the Azure IoT edge security daemon in Rust. [Online]. Available: https://msrc-blog.microsoft.com/2019/09/30/building-the-azure-iot-edge-security-daemon-in-rust/

[7] D. Barsky, A. Gupta, and J. Peddicord. (2019) AWS' sponsorship of the Rust project. [Online]. Available: https://aws.amazon.com/blogs/opensource/aws-sponsorship-of-the-rust-project/

[8] D. Herman. (2016) Shipping Rust in Firefox. [Online]. Available: https://hacks.mozilla.org/2016/07/shipping-rust-in-firefox/

[9] D. Kitchen. (2019) How we made firewall rules. [Online]. Available: https://blog.cloudflare.com/how-we-made-firewall-rules/

[10] S. Jayakar. (2020) Rewriting the heart of our sync engine. [Online]. Available: https://dropbox.tech/infrastructure/rewriting-the-heart-of-our-sync-engine

[11] J. Howarth. (2020) Why Discord is switching from Go to Rust. [Online]. Available: https://blog.discord.com/why-discord-is-switching-from-go-to-rust-a190bbca2b1f

[12] K. Ferdowsi. (2022) The usability of advanced type systems: Rust as a case study. [Online]. Available: https://weirdmachine.me/papers/usability_of_advanced_type_systems.pdf

[13] W. Crichton, "The usability of ownership," in *Proceedings of Human Aspects of Types and Reasoning Assistants (HATRA '20)*, 2020.

[14] A. Zeng and W. Crichton, "Identifying barriers to adoption for Rust through online discourse," *CoRR*, vol. abs/1901.01001, 2019. [Online]. Available: http://arxiv.org/abs/1901.01001

[15] 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 42nd International Conference on Software Engineering, ICSE*, 2020.

[16] M. Coblenz, M. L. Mazurek, and M. Hicks, "Garbage collection makes Rust easier to use: A randomized controlled trial of the Bronze garbage collector," in *International Conference on Software Engineering (ICSE)*, 2022.

[17] W. Crichton, M. Agrawala, and P. Hanrahan, "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, 2021, pp. 56:1–56:13. [Online]. Available: https://doi.org/10.1145/3411764.3445257

[18] J. J. Canas, M. T. Bajo, and P. Gonzalvo, "Mental models and computer programming," *International Journal of Human-Computer Studies*, vol. 40, no. 5, pp. 795–811, 1994.

[19] J. M. Carroll and J. R. Olson, "Mental models in human-computer interaction," *Handbook of human-computer interaction*, pp. 45–65, 1988.

[20] J. Urquiza-Fuentes and J. Á. Velázquez-Iturbide, "A survey of successful evaluations of program visualization and algorithm animation systems," *ACM Transactions on Computing Education (TOCE)*, vol. 9, no. 2, pp. 1–21, 2009.

[21] 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.

[22] E. E. Fırat and R. S. Laramee, "Towards a survey of interactive visualization for education," *Proc. Computer Graphics and Visual Computing*, pp. 91–101, 2018.

[23] P. J. Guo, "Online Python tutor: embeddable web-based program visualization for CS education," in *Proceedings of the 44th ACM Technical Symposium on Computer Science Education*, 2013, pp. 579–584.

[24] P. Caserta and O. Zendra, "Visualization of the static aspects of software: A survey," *IEEE transactions on visualization and computer graphics*, vol. 17, no. 7, pp. 913–933, 2010.

[25] O. Chitil, "Compositional explanation of types and algorithmic debugging of type errors," in *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming (ICFP '01), Firenze (Florence), Italy, September 3-5, 2001*, B. C. Pierce, Ed. ACM, 2001, pp. 193–204. [Online]. Available: https://doi.org/10.1145/507635.507659

[26] K. Tsushima and K. Asai, "An embedded type debugger," in *Implementation and Application of Functional Languages - 24th International Symposium, IFL 2012, Oxford, UK, August 30 - September 1, 2012, Revised Selected Papers*, ser. Lecture Notes in Computer Science, R. Hinze, Ed., vol. 8241. Springer, 2012, pp. 190–206. [Online]. Available: https://doi.org/10.1007/978-3-642-41582-1_12

[27] H. Simoes and M. Florido, "TypeTool: A type inference visualization tool," in *WFLP'04: Proc. 13th Intl. Workshop on Functional and (Constraint) Logic Programming, H. Kuchen, Ed. RWTH Aachen, Dept. Comp. Sc., Technical report AIB-2004-05*. Citeseer, 2004, pp. 48–61.

[28] T. Hill, J. Noble, and J. Potter, "Visualising the structure of object-oriented systems," in *Proceeding 2000 IEEE International Symposium on Visual Languages*. IEEE, 2000, pp. 191–198.

[29] ——, "Scalable visualisations with ownership trees," in *Proceedings 37th International Conference on Technology of Object-Oriented Languages and Systems. TOOLS-Pacific 2000*. IEEE, 2000, pp. 202–213.

[30] ——, "Scalable visualizations of object-oriented systems with ownership trees," *Journal of Visual Languages & Computing*, vol. 13, no. 3, pp. 319–339, 2002.

[31] D. Clarke, J. Östlund, I. Sergey, and T. Wrigstad, "Ownership types: A survey," *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*, pp. 15–58, 2013.

[32] P. Ruffwind. (2017) Graphical depiction of ownership and borrowing in Rust. [Online]. Available: https://rufflewind.com/2017-02-15/rust-move-copy-borrow

[33] J. Walker. (2019) Rust lifetime visualization ideas. [Online]. Available: https://blog.adamant-lang.org/2019/rust-lifetime-visualization-ideas/

[34] P. D. Faria. (2019) Borrow visualizer for the Rust language service. [Online]. Available: https://internals.rust-lang.org/t/borrow-visualizer-for-the-rust-language-service/4187

[35] D. Dominik, "Visualization of lifetime constraints in Rust," 2018. [Online]. Available: https://ethz.ch/content/dam/ethz/special-interest/infk/chair-program-method/pm/documents/Education/Theses/Dominik_Dietler_BA_report.pdf

[36] D. Blaser, "Simple explanation of complex lifetime errors in Rust," 2019. [Online]. Available: https://ethz.ch/content/dam/ethz/special-interest/infk/chair-program-method/pm/documents/Education/Theses/David_Blaser_BA_Report.pdf

[37] W. Crichton, M. Patrignani, M. Agrawala, and P. Hanrahan, "Modular information flow through ownership," in *PLDI '22: 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, San Diego, CA, USA, June 13 - 17, 2022*, R. Jhala and I. Dillig, Eds. ACM, 2022, pp. 1–14. [Online]. Available: https://doi.org/10.1145/3519939.3523445

[38] D. Miedema and G. Fletcher, "SQLVis: Visual query representations for supporting SQL learners," in *2021 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, 2021, pp. 1–9.

[39] A. Kaplan and D. Shoup, "CUPV—a visualization tool for generated parsers," *SIGCSE Bull.*, vol. 32, no. 1, p. 11–15, mar 2000. [Online]. Available: https://doi.org/10.1145/331795.331801