The Usability of Advanced Type Systems: Rust as a Case Study

KASRA FERDOWSI, UC San Diego, USA

Advanced type systems that enforce various correctness and safety guarantees—such as linear and ownership types—have a long history in the Programming Languages research community. Despite this history, a human-centered evaluation of these type systems and their usability was all but absent, with empirical evaluations limited to testing their expressiveness in programs written by experts, i.e. the creators of the type system.

In the past few years, this has begun to change with the adoption of a version of affine types and ownership in the popular Rust programming language. With the increase in Rust's popularity, various studies have begun empirically evaluating the usability of Rust's Ownership and Lifetime rules, providing a breadth of qualitative and quantitative information on the usability of such type systems. They found that despite Rust's general success in achieving its promise of safety and performance, these rules come with a steep learning curve and have been repeatedly cited as a barrier to adopting Rust.

In this report, I provide a brief history of linear types and region-based memory management, which directly inspired Rust's type system. I then introduce Rust's Ownership and Lifetime rules, and present the state-of-the-art in academic research into their usability. I discuss both theoretical arguments and empirical evidence for why these rules are difficult to learn and apply, and survey existing work on addressing some of these difficulties. I also draw from broader works in the HCI and CS Education communities to recommend future work in this area.

1 INTRODUCTION

Despite a plethora of work on advanced type systems in the Programming Languages research community, from Dependent types [Xi and Pfenning 1999] to Linear [Wadler 1990] and Ownership types [Clarke et al. 2013], such type systems have rarely crossed the academic boundaries into mainstream general-purpose programming languages.

A side-effect of this, perhaps exacerbated by a disinterest in human-centered methods in the Programming Languages community in the past [Coblenz et al. 2018], is the lack of any notable user evaluation of such type systems. So while their usability was repeatedly discussed, the focus was on whether any given type system is "expressive", i.e. can an expert write complex and useful code that type checks in that system. This meant that, until very recently, we simply did not know if such type systems are easy to learn, or how non-experts would learn and use them.

This has begun to change with the Rust programming language [Klabnik and Nichols 2017; Matsakis and Klock 2014]. Rust implement a notion of "Ownership", "Borrowing", and "Lifetimes" as a type system, which allows it to promise memory and thread safety at compile-time without garbage collection. And with its emergence as an increasingly popular general-purpose programming language, there has come a new wave of research into the usability of its Ownership model.

In this report, I aim to use this new research to better understand the usability of advanced type systems, and to see if and how they may be adopted by mainstream software engineers. The rest of the report is organized as follows: Sec. 2 provides a brief background in the history of the type systems most relevant to Rust. Sec. 3 then introduces Rust's specific implementation of those type systems, and Sec. 4 surveys the work on evaluating and improving the usability of Ownership in Rust. Finally, Sec. 5 combines takeaways from those works with theories from the Human-Computer Interactions and Computer Science Education communities to discuss possible next-steps in research on the usability of advanced type systems.

Research Exam, April 27 2022, La Jolla, CA, USA
2022.

2 BACKGROUND

This section provides a brief overview of Linear Types, Region-Based Memory Management (RBMM) and Ownership types, followed by a higher-level discussion of common themes among them. This discussion is not meant to be exhaustive, as each system has a long history and would require a separate survey paper by itself[1]. Instead, it is meant to provide a background for the theories that inspired Rust, and to help better connect the usability findings for Rust to type systems more broadly.

2.1 Linear Types

Inspired by Girard's Linear Logic [Jervell 1996], Linear Types were introduced by [Wadler 1990] as a way to safely "change the world" (i.e. modify state) in functional programming languages. The core of Linear Types are values that must be "used exactly once", i.e. they cannot be duplicated or implicitly discarded. Wadler pointed out that this restriction enables a number of static checks and features that may be very useful for memory management and program correctness.

More specifically, he noted that Linear Types enable memory management of mutable values without Garbage Collection. If a value cannot be copied or implicitly discarded and it must be used exactly once, then we can reclaim its memory after it is used. This handles memory management, and prevents use-after-free and double-free bugs. By prohibiting aliasing, Linear Types also solve the problem of reasoning about mutations, both in single-threaded code (where aliased references are a notable source of bugs), and in multi-threaded code (where aliasing can lead to race conditions).

In what will become a common theme in these type systems, Wadler also notes that strict Linear Types are a stronger constraint than necessary, and the language he introduces is not strictly Linear. Instead, it allows multiple "read accesses" (immutable references) to a Linear value that cannot be used once there is a "write" access (mutable reference). I will discuss Linear Types' relation to Rust more in Sec. 3, but Rust also uses a looser notion than Linearity called *Affine* types. Rather than being used *exactly* once, a value with an affine type must be used *at most* once, i.e. it can be ignored [Pierce 2004].

While I will not cover them in depth here, it is worth mentioning that Linear Types have been implemented and extended in various works, from their implementation in the imperative programming language Vault [Fahndrich and DeLine 2002] to their recent addition to Haskell [Bernardy et al. 2017]. They have also been an inspiration for the rest of the type systems in this section. Linearity is cited in both the seminal works on Region-based Memory Management [Tofte and Talpin 1997] and Ownership Types [Clarke et al. 1998], and has affected them over time, with [Walker and Watkins 2001] combining it with Region types, and the notion of Ownership transfer combining linear and non-linear types [Clarke et al. 2013].

2.2 Region-based Memory Management

As the name suggests, Region-based Memory Management (RBMM) was an effort in static memory management using the type system. RBMM started as an extension of Effect Type Systems [Pierce 2004] in [Tofte and Talpin 1997]. But its implementation for the Cyclone language [Grossman et al. 2002a,b; Jim et al. 2002] is the more direct influence on Rust, and so I will focus on that here.

Cyclone began as a part of the Typed Assembly Language project [Morrisett et al. 1999], but was developed into a separate project aiming to become "a safe dialect of C" [Grossman et al. 2005]. As such it contains a number of interesting design choices and language features besides RBMM such as tagged unions, null checks, and existential types. [Grossman et al. 2005] offers a

[1]Which actually exists in the case of Ownership Types [Clarke et al. 2013]

concise description of all these features, but here I will focus on RBMM in particular as described in [Grossman et al. 2002a], as it is the most relevant to this report.

The key idea of RBMM is to associate a lexically-scoped part of the program with a named "region" (a dynamically growable part of memory), and annotate the type of pointers into that region with the region's name. Then, the compiler can automatically deallocate the entire region at the end of the scope, and the type-checker can guarantee that pointers into a region are not dereferrenced outside of that region's scope (i.e. after it is deallocated). To do this, the type system keeps track of the set of regions that are live at each point in the program (called the "capability" at that point), and prohibits pointer dereferencing unless that pointer's associated region is in the capability.

This is the core of RBMM, but to make it sufficiently expressive and guarantee soundness, there's a lot more subtlety involved. For example, region types can support *subtyping*. Since regions can be nested, all pointers into the outer region are guaranteed to be alive during the inner one (since the outer region is deallocated after the inner). So if a region `'a` contains a smaller region `'b`, pointers annotated with `'a` are a subtype of `'b`.

Another detail that Cyclone developers considered was the syntactic overhead of region annotations, and the need for region generics (functions with arguments and return types that are generic over region annotations). Their solution was a combination of intraprocedural region annotation inference, which removed the need for most explicit annotations in function bodies, and defaults for partially-annotated function signatures. This removed a large amount of the syntactic overhead, and made certain functions translate from C to Cyclone directly with no or minimal change.

Outside of Cyclone, RBMM has had a long history. It has been implemented for the Go programming language [Davis et al. 2012], Real-Time Java [Boyapati et al. 2003b], Prolog [Makholm 2000], GPU programming [Holk et al. 2014], and Big Data systems [Nguyen et al. 2015]. But none of these works empirically evaluated the usability of their system on programmers, focusing instead on benchmark performance and limiting their discussion of usability to expressiveness and syntactic overhead.

2.3 Ownership Types

Despite having a similar name, Ownership Types are not directly related to Rust's notion of Ownership. However, they share many of their goals with Rust, and are a key part of the history of Ownership as a concept. So a discussion of relevant type systems for Rust would be incomplete without them.

Ownership Types were developed as a part of Object-Oriented Programming (OOP) to statically enforce a more strict notion of "encapsulation" [Clarke et al. 2013]. While the details vary greatly between implementations, the general idea is to encode an "owning" and "owner" relationship between objects in the type system. This places restrictions on pointer aliasing which enforce encapsulation [Clarke et al. 1998], and enable additional checks and guarantees, such as preventing data races and deadlocks [Boyapati et al. 2002] and dangling pointers [Boyapati et al. 2003b].

Despite the large body of work on Ownership types, and its close relation to Java (a popular general-purpose programming language), these types were neither widely adopted, nor evaluated on real users. Instead, each extension, implementation or application of these types was only evaluated by the designers of the system, who programmed real-world applications with their type system to argue for its expressiveness [Aldrich et al. 2002; Boyapati et al. 2003a,b; Clarke et al. 2013, 1998].

Despite having a mostly separate history, Ownership Types are conceptually very close to the other type systems in this report. For instance, in Ownership types "a program's heap is divided into hierarchically nested regions, originally called ownership contexts" [Clarke et al. 2013] which is similar to regions in RBMM. In fact, [Boyapati et al. 2003b] combined Ownership types with Region-based Memory Management to implement a type system for Real-Time Specification for Java (RTSJ). This system statically guaranteed the success of runtime checks for dangling pointers, and a lack of references to the garbage-collected heap (a requirement in RTSJ code). However, similar to the rest of the works in this section, this system was never evaluated on real users.

2.4 Shared Themes

So far I introduced each type system individually, but these ideas and systems are closely related, and their development is not easily separable. So, before moving to Rust, I will first discuss the broad trends in these works more holistically.

Linear, Region and Ownership Types are related by their attention to memory management and safety. Each realized that type systems could be used to help programmers reason about complex programs, prevent various errors in using aliased or freed references, and offer a provably correct solution to memory management without the need for runtime checks or garbage collection.

They were all also quick to note and try to tackle the trade-off between the "expressiveness" of their type systems, and their guarantees. In the paper that introduced Linear Types, [Wadler 1990] did not enforce Linearity, but allowed combining values of Linear and non-Linear types, and (as discussed above) loosened the definition of uniqueness to allow multiple read-only references to Linear values. Despite touting RBMM, Cyclone [Grossman et al. 2002b] included a distinguished garbage-collected heap region. And a few years after its introduction, early proponents of Ownership Types were already making the case that strict uniqueness is needlessly restrictive [Clarke and Wrigstad 2003].

Finally, despite this attention to expressiveness, a tendency to implement their type systems as versions or extensions of popular programming languages (ML [Tofte and Talpin 1997], C [Grossman et al. 2002b], Java [Aldrich et al. 2002], Scala [Haller and Odersky 2010], Go [Davis et al. 2012], Prolog [Makholm 2000], etc.), and even considering the benefit of such restrictions in program comprehension [Aldrich et al. 2002; Clarke et al. 2013], a user-centered approach was missing from all the works cited above. No one studied if users other than those who had invented and implemented the type systems could easily work with the restrictions imposed by them.

3 THE RUST PROGRAMMING LANGUAGE

Rust, though inspired by [Wadler 1990] for its notion of Ownership, and [Grossman et al. 2002b] for its approach to lifetimes and memory management, does not directly implement any of the type systems above. Instead, it combines them with a number of other ideas to try to guarantee memory- and thread-safety, as well as static memory management without garbage collection. However, it also aims to be a general-purpose systems programming language[2], and so aims for "pragmatic safety" [Evans et al. 2020] and has features that bypass its static checks for better performance or more complex aliasing patterns.

In the rest of this section, I will first introduce Ownership as it is used in Rust, then describe lifetimes and how they combine with Ownership, and finally describe unsafe code which bypasses some of these checks. I have restricted my descriptions here to what's necessary to think about some of the usability issues I will discuss in Sec. 4, and I am ignoring many subtleties of the type system, as well as any mention of Rust's syntax, semantics, etc. For a good introduction to Rust in

[2]The term "Systems Programming Language" has caused some controversy recently [Crichton 2018]. But given its colloquial use as a language that compiles to assembly, and offers low-level control of resources (as opposed to interpreted languages like Python, or those that run on higher levels of abstraction such as Java), I will use that term in this report for simplicity's sake.

```
1 let v = vec![1, 2];
2 let v2 = v;
3 print(&v);
```

(a)

```
1 let v = vec![1, 2];
2 let x = &v[0];
3 let v2 = v;
4 let y = *x + 1;
```

(b)

```
1 let mut v = vec![1, 2];
2 let x = &v[0];
3 Vec::push(&mut v, 3);
4 let y = *x + 1;
```

(c)

Fig. 1. Each subfigure demonstrates a violation of the corresponding rule in Sec. 3.1. In Fig. 1a ownership of the vector is transferred to `v2` on line 2, so `v` cannot be borrowed for the call to `print` on line 3. In Fig. 1b `x` is a reference to `v` and lives until its last use on line 4. But `v` only lives until the transfer of its vector to `v2` on line 3. Similarly, in Fig. 1c, the *immutable* borrow of `v` in `x` lives from line 2 to its last use on line 4, so `v` cannot be *mutably* borrowed by the call to push on line 3.

general, I recommend the official Rust book [Klabnik and Nichols 2017], which is both thorough and very readable.

3.1 Ownership

Ownership rules in Rust are, on paper, quite simple, and various papers have attempted to summarize them [Fulton et al. 2021; Qin et al. 2020]. Here I will use [Crichton 2020], since it is the most simple and concise. But first, I need to introduce some terms.

In Rust, each value (a `String`, `i32`, `Vec`, etc.) is owned by a single variable, which is its **owner**. Since working with values directly can be inconvenient, Rust also has **references** to values which **borrow** the value from its owner. Finally, in Rust variables and references are **immutable** by default. To mutate a value through one, it needs to be explicitly marked as **mutable** using the `mut` keyword. For instance, in Fig. 1c line 1, the `Vec` created by the call to `vec!` is assigned to the variable `v`, thus `v` owns that `Vec`. On line 2, `x` is a reference to the first element of `v`, and thus `x` borrows `v`. Finally, `v` is marked with the `mut` keyword, and thus it is mutable, but `x` is not, and so `x` can only be read, not modified or reassigned.

With ownership, references and mutability in mind, [Crichton 2020] summarizes Rust's Ownership rules like so:

(a) All values have exactly one owner.
(b) A reference to a value cannot outlive the owner.
(c) A value can have one mutable reference or many immutable references.

See Fig. 1 for a code example of what violating each of these rules may look like. Rust's compiler `rustc` contains a pass, known colloquially as the borrow-checker, which fails if it cannot statically determine that all of these rules are being followed.

Before moving on to lifetimes, it is worth considering how these rules relate to the type systems in Sec. 2. Rather unintuitively, Rust's Ownership rules are not directly related to Ownership Types discussed in Sec. 2.3[3]. Rather, the first rule which is generally referred to as "Ownership" is most closely tied to Linear Types: A value has a single owner at any point in the program, and while Ownership can be transferred between variables, it cannot be implicitly duplicated.

This is not to say that Ownership Types are entirely unrelated. Mutability and borrowing are more explicitly dealt with in Ownership Types than Linear or Region types. Certain instances of Ownership Types restrict mutation to the owner of a value, and only permit read-access from other objects. Similarly, the notion of borrowing appears in parts of the Ownership Type literature with a similar function [Aldrich et al. 2002; Boyland 2001]. But Ownership Types are closely tied

[3]This confusion is further exacerbated by more recent papers such as [Crichton et al. 2021] which refer to Rust's Ownership rules as "Ownership Types", despite the existing history of Ownership Types as a different system.

```
struct Foo<'small, 'large: 'small> {
    a: &'small str,
    b: &'large str,
}
impl<'s, 'l: 's> Foo<'s, 'l> {
    fn new(x: &'s str, y: &'l str) -> Self {
        Self { a: x, b: y }
    }
}
```

```
fn main() {
    let mystr = "abc";
    let substr = &mystr[0..2];
    let foo = Foo::new(&mystr, substr);
}
```

Fig. 2. An example of explicit lifetime parameters in **struct** and **fn** definitions. The lifetime parameters define the **struct** and **fn** as generic over lifetimes `'small` and `'large`, where `'large` is a subtype of `'small`. Rust then uses these parameters to type check the use of `Foo::new` in the `main` function by comparing the inferred lifetimes for the references passed to `Foo::new` with the explicit lifetime parameters. This code compiles because `substr` borrows `mystr` and so its lifetime must be smaller than `mystr`, which satisfies the subtyping requirement between `'small` and `'large`.

to concepts from Object-Oriented Programming (which is not Rust's paradigm). And I have only found a single mention of Ownership Types as an influence on Rust in the literature [Weiss et al. 2021].

I will leave a detailed comparison to RBMM to the next section, but note how rules (a) and (b) also allow automatic memory management: When the owner of a value goes out of scope, it is guaranteed not to have any live references, and so the compiler can insert a call to deallocate that value (`drop` the value in Rust parlance). This preserves memory-safety without the need for garbage collection.

3.2 Lifetimes

[Fulton et al. 2021] gives a great and concise description of lifetimes:

> "A lifetime names a scope, and a lifetime annotation on a reference tells the compiler the reference is valid only within that scope."

Lifetime annotation here refers to the fact that Rust references are not the same as C-style "raw" pointers. A raw pointer's type only has the type of the value it points to. But the type of a Rust reference is annotated with a lifetime that refers to the scope where that reference is valid, and lets the borrow-checker keep track of which value (or other reference) it is borrowing.

Rust automatically infers all lifetimes in function bodies, and so most annotations are not visible to the user. However, when writing functions that have references in their signatures, or data types which store references, Rust requires users to explicitly write them as generic functions/data types over the lifetimes of those references[4]. You can see examples of this in Fig. 2.

There is much more to lifetimes, how they are calculated, and their implications on expressiveness and usability. But, as we shall see, Rust lifetimes are notoriously complex and difficult, and a full description of these aspects is outside the scope of this report. So I will leave lifetimes here, and end this section with a discussion of their relation to Cyclone.

As the reader may have noticed by now, lifetimes in Rust are very similar to regions and region annotations in Cyclone, including their syntax, subtyping, and intraprocedural inference. In fact,

[4]There is an exception to this, which is functions whose signatures follow a particular pattern such as functions that don't return a reference or which take exactly one reference and return a reference. In these cases, Rust *elides* these lifetimes as a syntactic convenience. Note that this is not the same as lifetime inference. Rust, similar to Cyclone, does *not* infer lifetimes in function signatures.

many of Rust's features were inspired by Cyclone, such automatic bounds checking, and sum types (Rust `enum`s and Cyclone's tagged unions).

The main difference between the two is that Cyclone requires manually defining the syntactic scope of regions, and using the region names (which are first-class values) to manually allocate and initialize values inside different regions. In Rust, lifetimes are automatically determined by the compiler, and cannot be explicitly set or used (except for generic lifetime parameters). Also, since Rust 2018, Rust regions are not determined lexically, but are instead calculated over an intermediate control-flow graph representation of the program [The Rust Core Team 2018]. So while the theory behind regions and lifetimes is the same, their implementation and interaction model are different in interesting and significant ways.

3.3 Unsafe Rust

One of the common themes among the type systems discussed in Sec. 2 was that each found strict adherence to its rules needlessly restrictive, and found ways to loosen it for the sake of expressiveness. Rust is no exception to this, though its solution is rather different.

An important issue with Rust's Ownership rules is that they are sound, but undecidable. So the borrow-checker is incomplete, and there is plenty of safe code which follows the Ownership rules, but the borrow-checker cannot statically verify. To alleviate this, Rust allows users to explicitly mark functions and blocks of code as `unsafe`, and in these unsafe blocks, certain safety checks are disabled[5]. More specifically, `unsafe` allows the code to:

- Dereference raw pointers
- Call unsafe functions (including C functions, compiler intrinsics, and the raw allocator)
- Implement unsafe traits
- Mutate statics
- Access fields of unions

[Rust Project Developers 2022]. This is still quite restrictive, but has serious implications. For instance, dereferencing raw pointers can work around the Ownership rules by casting a reference with one lifetime into a raw pointer, and dereferencing that raw pointer to borrow it again as a new reference *with a new lifetime*. This allows creating multiple mutable references to a value at the same time, which violates the third rule of Ownership.

The details of this are again notoriously complex and beyond the scope of this report, but unsafe code is crucial to Rust's "pragmatic safety". It allows various performance improvements that are too low-level for the borrow-checker to reason about, as well as interfacing with external code, and certain aliasing patterns which the programmer can verify as safe, but do not pass the borrow-checker.

4 THE USABILITY OF OWNERSHIP

At the time of writing, there have been five major papers on the usability of Rust's Ownership type system [Coblenz et al. 2021; Crichton 2020; Fulton et al. 2021; Zeng and Crichton 2019; Zhu et al. 2022]. And, following the best practices of behavioral research [Mcgrath 1995], they use a variety of methods to inspect a number of similar and overlapping research questions. So, rather than review each paper individually, in this section I discuss their collective findings, introducing the papers and their methodology as they become relevant. I also draw from research on the use of `unsafe` in Rust, as well as related research on how programmers learn a new programming language.

[5]Some, [Zhu et al. 2022] for example, refer to unsafe code as "similar to the C Programming Language". While this is technically true because unsafe code can interface with arbitrary C code, unsafe code within Rust is still far more restrictive than C.

The high-level takeaway is that Rust's Ownership model is indeed difficult to learn, and certain aspects of its design remain difficult even for more experienced Rust developers. However, its promise of safety and performance, coupled with good tooling and features for interoperability with other languages, keep Rust popular and loved by those who succeed in adopting it.

4.1 Is Ownership difficult to use?

> "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."
>
> — Student participant from [Coblenz et al. 2021]

Rust is notorious for it's "steep learning curve", and this has been noted as a major issue in adopting it in the industry. But interestingly, studies suggest that even experienced developers struggle with certain aspects of Ownership in Rust.

4.1.1 Barriers for Novice Rust Programmers. [Zeng and Crichton 2019] performed content analysis on top posts from the `/r/rust` subreddit (an online community specific to Rust), and articles and corresponding comments from Hacker News (a broader tech forum) to identify barriers to adopting Rust. In 18 experience reports and language comparisons they inspected, they found that "the complexity of the borrow-checker was the second most frequently mentioned complaint" (second only to compiler version issues), where memory access patterns that were common in other languages were disallowed by the borrow-checker, leading to frustration.

[Fulton et al. 2021] followed this work by interviewing 16 industry professionals who had attempted to adopt Rust in their production team, and used their findings to design an online survey which provided them with 178 more participants. They also found that Rust's steep learning curve was the most serious barrier to adoption.

Note that this difficulty is more than simply the difficulty of learning a new language, or indeed learning a systems programming language without garbage collection. [Fulton et al. 2021] found that the biggest challenge in learning Rust was specifically the borrow-checker, and the necessary shift in programming paradigm to write code that passes the borrow-checker. And [Shrestha et al. 2020] quoted a C++ developer who said that the borrow-checker "forces a programmer to think differently". So it appears that Rust's more advanced type system is the main source of its difficulty, not just a lack of garbage collection, or more low-level programming.

Which is not to say that adding garbage collection will not ease Rust's difficulty. [Coblenz et al. 2021] ran a controlled study on 428 students in a sophomore-level programming course. The students were given two weeks of lectures on Rust, and then asked to complete an assignment which required a good understanding of Rust, its Ownership rules, and types that allow for interior mutability. They randomly assigned students to two groups, one having to complete the assignment using the Rust standard library data types, and one using a garbage-collected wrapper type (called "Bronze") which enabled a number of additional aliasing patterns to pass the borrow-checker, thus removing the needs for more complex aliasing patterns and datatypes.

They found a significant difference in the rate of completion and the self-reported time to completing the assignment. The students who used Bronze on average took only a third as much time as the control group, and were approximately 2.44 times more likely to complete the assignment. Interestingly, the time difference between the groups only appeared in the second part of the task, which involved complex aliasing and mutability. The first part of the assignment, which focused just on Ownership, didn't show a significant difference between the groups.

4.1.2 Barriers for Experienced Rust Programmers. Aside from the initial learning curve, studies also suggest that aspects of Ownership remain difficult to use, even for experienced developers. In their study of memory- and thread-safety issues in Rust, [Qin et al. 2020] inspected five Rust systems and applications, five popular libraries, and two vulnerability databases. They found that a common reason for blocking bugs in these codebases was a lack of "good understanding in Rust's lifetime rules". This is notable since, unlike the participants in the studies above, the programmers who worked in these codebases were presumably experienced Rust developers.

This finding is corroborated by results from the Rust community's 2020 survey [The Rust Survey Team 2020]. They received 8323 responses, with the largest number of participants self-reporting their expertise as 7 out of 10. They also found that lifetimes are the most difficult topic to learn, though unfortunately they do not report if and how this response changes according to the expertise rating. Similarly, in their study of 100 samples of StackOverflow questions on Rust's Ownership rules, [Zhu et al. 2022] found that the most common cause of safety rule violations in these questions was "complex lifetime computation", which appeared 74 times[6], 44 in intraprocedural lifetime computation, 16 in explicit lifetime parameters, and 14 in elided ones.

From these works, it seems safe to conclude that Rust's Ownership rules are indeed difficult to learn. They pose a serious barrier to learning and adopting Rust, and understanding lifetimes specifically remains a problem even for more experienced Rust developers.

4.2 Why is Ownership difficult to use?

> "I can teach the three rules [of Ownership] in a single lecture to a room of undergrads. But the vagaries of the borrow checker still trip me up every time I use Rust!"
>
> — [Crichton 2020]

If we accept that Rust is indeed more difficult to learn than comparable systems programming languages, and that this is in large part caused by its Ownership type system specifically, the next step is to ask what about Rust's Ownership rules is difficult to learn and apply.

4.2.1 Change of Paradigm. One answer may be the notion of "interference" as used by [Shrestha et al. 2020]. In that paper, they qualitatively coded 450 posts on StackOverflow for 18 different programming languages, and interviewed 16 professional programmers, to understand how experienced developers learn new programming languages, and what they struggle with in the process. They motivated this work by borrowing the term "interference"[7] from psychology and neuroscience. The term refers to when "previous knowledge disrupts recall of newly learned information". This can be as simple as the difference in zero- vs. one-indexing between two languages, but it also applies to larger differences, where programming in the new language requires a "mindshift", or a fundamental change in paradigms.

Learning Rust needs such a mindshift, because its Ownership rules prohibit many common programming patterns. Consider a doubly-linked list. In most languages its implementation is close to trivial, but it violates Rust's rules by definition: It requires at least two mutable references to a node, one from the previous and one from the next node. Rust has workarounds for this, most simply datatypes with "interior mutability" that postpone checking for simultaneous mutable access to runtime, but they are more difficult to learn and work with. So it is unsurprising that [Shrestha et al. 2020] use Rust's Ownership type system as an example of mindshifts, quoting a C# developer who had to "completely rethink the problems they would have normally solved in C#".

[6] The paper counts 77 violations, but I'm exlcluding 3 which were merely syntax errors.
[7] As well as the term "facilitation", but that is not as relevant here.

```rust
let mut v = vec![1, 2];
let one = &mut v[0];
let two = &mut v[1];
*two += *one;
```

(a)

```rust
let mut v = vec![1, 2];
let iter = v.iter_mut();
let one = iter.next().unwrap();
let two = iter.next().unwrap();
*two += *one;
```

(b)

```rust
1 let mut v = vec![1, 2];
2 v.insert(0, v[0]);
3 v.get_mut(v[0]);
4
5
```

(c)

Fig. 3. The programs in Fig. 3a and Fig. 3b perform the same function, but only Fig. 3b passes the borrow-checker. In Fig. 3c, the statements on lines 2 and 3 are nearly identical at the type-level, but only line 2 passes the borrow-checker, presumably due to some implementation detail.

In the qualitative portion of their study, [Coblenz et al. 2021] noted a similar theme in the students' survey responses. For the students without the Bronze library, the second part of the assignment required using types with interior mutability and explicit lifetime parameter declarations. Students mentioned the difficulty of using these types, and the need for redesigning their code to use them correctly, leading the authors of the paper to conclude that "most of the benefit of GC comes from architectural simplification" and that "design was a significant contributor to the difference in performance between non-Bronze and Bronze participants."

So at least one main reason for the difficulty of learning Ownership is that it requires a change of paradigm. A programmer who is new to Rust needs to learn entirely new patterns and ways of structuring code at the architectural level. And their previous experience can actively interfere with their learning, as they need to abandon common programming patterns and learn to structure their code in new and unintuitive ways.

4.2.2 Error Messages. [Coblenz et al. 2021] also noted that `rustc`'s error messages contributed to the confusion and frustration. `rustc` error message not only describe the error in the code, but for certain error patterns, suggest edits that may fix the problem. However, these edits are always local and don't provide any high-level design feedback which may be helpful in making the mindshift. At best, they led the students to perform a chain of local edits that resulted in code that compiles without them understanding why. At worst, as one student found, they could be cyclical "with things like `remove &` then after removing `try adding &`." This lead the authors to conclude that Rust's error messages do not "aid design or comprehension".

[Zhu et al. 2022] investigated error messages more directly. They employed Cognitive Task Analysis [Diaper 2004] to learn how experts solve 110 Rust Ownership errors they had identified in a sample of StackOverflow questions, and compared the steps the experts took to the information contained in the error message. They found that while for most errors the error message contained all relevant information, for 32 errors the message failed to explain "the key steps in computing a lifetime or a borrowing relationship", with another 10 failing to "explain the relationship between two lifetime annotations", and 9 "how a safety rule works on a particular code construct".

I will leave a broader discussion of `rustc` error messages to Sec. 5, but the works cited here indicate that Rust error messages do not provide the necessary help. Programmers' errors may be more structural, but the error messages only suggest potentially misleading local edits. And even for local errors, they do not always contain the necessary information to understand and fix the error, and assume external knowledge on behalf of the programmer. But this still doesn't explain why an *experienced* Rust developer struggles with Ownership.

4.2.3 The Curse of Incompleteness. [Crichton 2020] point out that Ownership rules are simple and easy to learn, but statically checking for them, "like most interesting program properties", is undecidable. So Rust's implementation of these rules in the borrow-checker is necessarily incomplete, and a lot of the usability issues with Ownership come from this gap between the programmer's understanding of the rules, and the borrow-checker's ability of verify them.

Consider the examples in Fig. 3a and Fig. 3b. Both programs perform a similar function, getting references to two elements of a vector and incrementing one by the other. But only the second compiles, since the borrow-checker cannot reason about indices. It conservatively assumes that both lines 2 and 3 in Fig. 3a are mutably borrowing the entire vector, thus violating Rule (c) in Sec. 3.1. Fig. 3b, however, uses an iterator, which the borrow-checker can reason about at the type level[8], and can successfully verify does not violate the Ownership rules. Thus Fig. 3b compiles successfully. Note that both of these programs are "safe", and a more advanced type system involving dependent types could in theory statically verify the safety of Fig. 3a, but the current limitations of the type system means that developers need to learn, not just the rules of Ownership, but how the borrow-checker verifies them.

This issue is exacerbated by the fact that the borrow-checker's implementation is quite complex and sometimes very similar code may not compile for obscure reasons. The code example in Fig. 3c has two similar calls to functions on the vector. Line 2 gets the first element of `v`, and inserts it as the new first element of `v`. Line 3 uses the *value* of the first element as an index to get a mutable reference to the second element of `v`. These functions have very similar types, both using an immutable borrow of `v` to get an argument for a call that mutably borrows `v`. However, as of Rust version `1.59.0`, line 2 compiles successfully, but line 3 fails the borrow-checker[9].

There are almost certainly many other large and small, obvious and subtle reasons for the difficulty of learning and using Rust's Ownership type system, but these three (Rust's different paradigm, unhelpful error messages, and the incompleteness of the borrow-checker) are the most apparent from the works surveyed here.

4.3 Why do developers try to use Rust anyway?

> Instead of having to invoke `pkg-config` by hand or with Autotools macros, wrangling include paths for header files and library files and basically depending on the user to ensure that the correct versions of libraries are installed, you write a `Cargo.toml` file which lists the names and versions of your dependencies. [...] It just works when you `cargo build`.
>
> — [Mena-Quintero 2018][10]

The last question I will inspect here is that, if Ownership is difficult to learn and use for so many reasons, why do developers choose to use Rust anyway?

And perhaps the first answer to that is that they don't. Despite being the "Most Loved" programming language in every StackOverflow survey since 2016 [Stack Overflow 2016, 2017, 2018, 2019, 2020, 2021], it's user-base is small and growing slowly. In the same surveys, it appeared in the list of Most Popular languages in 2019 at only 3.2% [Stack Overflow 2019], growing to 7.03% in the latest survey [Stack Overflow 2021]. In comparison, the Go programming language (which is often compared with Rust as a modern systems programming languages) was already at 8.2% in 2019, though it only grew to 9.55% by 2021. Similarly, the TiOBE index ranks Rust at 26 [TIOBE Software BV 2022], and the IEEE Spectrum ranks it at 17 [Cass et al. 2022], compared to 13 and 8 for Go.

---

[8]I should mention that `iter_mut` uses **`unsafe`** to achieve this under the hood, but since `iter_mut` is itself a safe function provided by the Rust standard library, it can easily be used by novices without ever touching unsafe code.

[9][Crichton 2020] speculates that the reason for this is that `get_mut` is defined on slices (which the `Vec` type implements), while `insert` is implemented directly on `Vec`. They don't know why this distinction matters, and it only further proves their point.

[10]Quote found in [Zeng and Crichton 2019].

There could be many reasons for this beyond the usability of Ownership of course, and these are not peer-reviewed sources.

But Rust's popularity *is* still growing, and unsurprisingly the main reason most participants in multiple studies cited was its promise of memory- and thread-safety [Fulton et al. 2021; Zeng and Crichton 2019]. Unfortunately, neither of these papers go into depth about this, and only mention that safety is the most commonly noted reason. However, other themes besides safety emerged in these works that are far more interesting.

The first is that while safety is important, it is not enough. [Zeng and Crichton 2019] noted that while the first and third most-noted benefits of adopting Rust were avoiding runtime errors and data races, the second most-mentioned benefit was Rust's build tool `cargo`, which avoided the many issues of build tools for other langauges. Similarly, while [Fulton et al. 2021]'s participants cited Rust's safety as a benefit most frequently, they listed performance and lack of garbage collection almost as frequently.

Another theme that came up in multiple works was Rust's **`unsafe`** feature. Two studies which inspected the use of **`unsafe`** in Rust code repositories found that unsafe code is common. [Evans et al. 2020] inspected all publicly available Rust libraries on `crate.io` (Rust's online library registry), and found that explicit **`unsafe`** blocks appear in 29% of all libraries. When they filtered their results to the most popular libraries (which accounted for 90% of downloads), this percentage increased to 52.5%. In the 5 applications they inspected, [Qin et al. 2020] found 4990 uses of **`unsafe`**, with a further 1581 unsafe code regions in the standard library, and concluded that unsafe code is used "extensively". Though they note that it is "unavoidable in many cases" and "usually for good reasons", including interfacing with existing libraries written in other unsafe languages such as C, and performance improvements by a factor of 4 or 5. Interestingly, they also found cases of the **`unsafe`** keyword being used as a warning to developers, despite the code itself being safe and compiling without the unsafe block. Those who tried to adopt Rust also noted the many uses for unsafe code, citing its necessity for integrating Rust into existing codebases through FFIs, accessing hardware, and for performance reasons [Fulton et al. 2021]. So it seems that "pragmatic safety" was an essential part of Rust's success, as a large amount of code written in Rust would have not been possible it if had strictly adhered to its statically-guaranteed safety rules.

It's also interesting to note that we now have empirical evidence that, despite **`unsafe`** being described as an "escape hatch" [Evans et al. 2020], developers can be trusted to use it responsibly, only where necessary and while still mostly maintaining Rust's safety guarantees. As [Evans et al. 2020] found, most Rust codebases do not contain explicit unsafe code. And in their study of StackOverflow questions, [Zhu et al. 2022] found that of the 110 errors in the questions, only 3 were fixed by writing unsafe code. The rest were either fixed with simple safe code, or library functions which used unsafe blocks internally, but exposed a safe interface (a programming pattern known as "interior unsafe"). In larger codebases, [Qin et al. 2020] listed numerous cases of memory bugs related to unsafe code, but they found that this has more to do with the complexity of the type system (and lifetimes especially), and not developer negligence. And good programming patterns such as interior unsafe, best-practices such as coding reviews, and better tools for reasoning about lifetimes and unsafe code could alleviate many causes for these bugs.

So while Rust is not as popular as comparable languages which lack its advanced type system, developers do like it and try to adopt it, mainly for its promise of safety. Though alongside safety, they also value its performance, lack of a garbage collector, and great tooling. And as much as they appreciate Rust's safety, they still have numerous justified reasons for using unsafe code, and do so responsibly.

5 FUTURE WORK

In this section, I will briefly introduce ideas from the fields of HCI and Computer Science Education that I believe could contribute to better understanding and improving the usability of Rust, or indeed any other advanced type system. These ideas are eclectic, and I mean to introduce them as a starting point for generating ideas, not as fully fleshed-out research plans. Also, as the focus of this report is on human-centered approaches, I will not discuss techniques from Programming Languages and Compilers research. Though, especially if combined with HCI techniques, they could be essential to improving the usability of Rust as well.

5.1 Better Error Messages

Perhaps the most immediately actionable takeaway from Sec. 4 is that Rust's error messages are an important limitation. But, paradoxically, the general community consensus is that Rust's error messages are more helpful than most languages [Fulton et al. 2021]. So a good next step in improving Rust's usability, and an important consideration when designing any language with such complex types, is to see where Rust's error messages succeed and how they fail.

A great starting point for this is [Becker et al. 2019]. They survey all works on compiler error messages in the past 50 years, and provide a number of remarkable insights on the subject. For one, they discuss empirical evidence showing that programmers, both novice and expert, *do* read compiler error messages [Barik et al. 2017; Prather et al. 2017], and so improving error reporting is indeed worthwhile.

They also compiled a list of empirically-backed guidelines for designing error messages, which can be invaluable as a shared foundation for synthesizing various works on Rust's usability. For example, it could serve as the complementary theoretical background to empirical analyses (such as in [Zhu et al. 2022]) that argue that Rust's error messages are notably well-designed[11].

It is also a good resource for understanding why Rust error messages fail, and how to improve them. For instance, one of the important guidelines for error design is to include the relevant context of the error directly in the message, which multiple works have argued Rust sometimes fails to do [Blaser 2019; Dominik 2018; Zhu et al. 2022].

Following [Coblenz et al. 2021]'s finding that Rust diagnostics lack necessary architectural hints, the guidelines would also be invaluable in providing the human-centered design element to programming languages or machine learning techniques that could identify such architectural changes and present them to the programmer.

5.2 Program Visualization Tools

An often mentioned next step for the usability of Rust's Ownership is visualization of Rust's lifetimes. In their discussion of ways to address the usability issues with the incompleteness of the borrow-checker, [Crichton 2020] recommended further research into visualizing the various static information provided by the borrow-checker, citing the difficulty in visualizing the large amount of information in a "succinct, non-intrusive, yet informative" manner.

Similarly, [Qin et al. 2020] suggested an IDE plugin for visualizing lock lifetimes. They found that a notably common cause of deadlock bugs in multi-threaded code was that in Rust unlocking a `Mutex` occurs implicitly when the locked value is dropped. To address this, they recommended "plug-ins to highlight the location of Rust's implicit unlock", which is a specialized case of visualizing lifetimes.

[11] A deeper look into this is outside the scope of this report, but my personal experience suggests that Rust follows most of the guidelines from [Becker et al. 2019] most of the time.

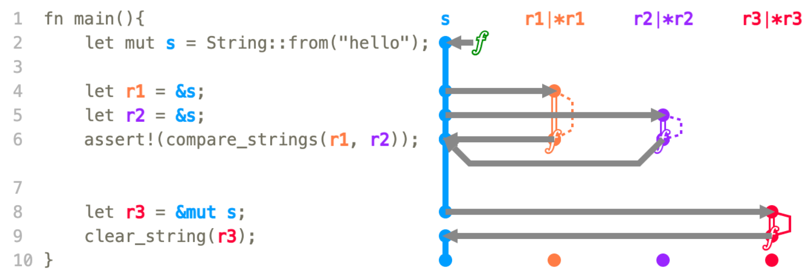

Fig. 4. A RustViz visualization. If the user hovers over the nodes and arrows on the right, it displays a pop-over containing additional information about the event.

There are three existing but limited works attempting to visualize Ownership and lifetimes in Rust. The first is RustViz [Gongming et al. 2020], a tool for building visualizations of Ownership and borrowing events in Rust programs. RustViz does not generate these visualizations automatically, but instead requires (arguably considerable) effort on the part of a "teacher" to specify them by annotating the code being visualized, and using a Rust library to generate it. The benefit is that the generated visualization is interactive: the "learner" can hover over parts of the image to get a brief description of the events. See Fig. 4 for an example of RustViz's visualizations.

The other two are related Bachelor's Theses [Blaser 2019; Dominik 2018]. In the first [Dominik 2018], the author uses Polonius (an experimental implementation of the borrow-checking rules in Datalog) to extract lifetime constraints from a given program, and visualize it as a directed graph. They motivated it by arguing that `rustc` error messages do not always present all the necessary information for Ownership errors, and that this visualization could address that problem.

[Blaser 2019] then extended [Dominik 2018] to improve its usability. They defined an algorithm which filters the complete graph to a single path that contains all constraints relevant to the particular error message, such as in Fig. 5. They also created a Visual Studio Code extension called "Rust Life Assistant" for displaying these graphs in the IDE, and an algorithm for converting the graph into a bullet-point list of English text explaining the cause of the error.

All these works are quite limited, and crucially none of them have been evaluated on users[12]. So a great next step could be to design, implement and evaluate a tool for visualizing Ownership and lifetimes. Such a tool could be pedagogical (similar to RustViz) or utilitarian (similar to Rust Life Assistant), but as [Zhu et al. 2022] points out "learning Rust is a continuous process", and I would recommend considering existing works on education and program visualization systems to inform the design and evaluation of any such tool.

A great survey of such systems can be found in [Sorva et al. 2013]. Besides exploring over 40 program visualization tools, they also provide a taxonomy of visualization systems, and emphasize that *how users engage* with a visualization tool is as important as the design of the tool itself. Specifically, they cite [Hundhausen et al. 2002] who performed a meta-study of 24 algorithm visualization (AV) tools and found that "the form of the learning exercise in which AV technology is used is actually more important than the quality of the visualizations produced by AV technologies." Unfortunately most tools focus on the runtime behavior and values of programs, but they can still help us better think about the design of a visualization system, and users' engagement with it.

[12] [Gongming et al. 2020] contains a study proposal, but as of writing this report, no results of such a study have been published.

```rust
fn main() {
    let mut x = 4;
    let y = foo(&x);
    let z = bar(&y);
    let w = foobar(&z);
    // ...
    x = 5;
    take(w);
}

fn foo<T>(p: T) -> T { p }
fn bar<T>(p: T) -> T { p }
fn foobar<T>(p: T) -> T { p }
fn take<T>(p: T) { unimplemented!() }
```

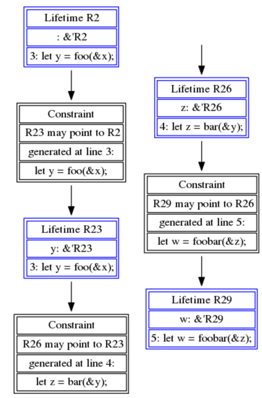

Fig. 5. A Rust Life Assistant visualization, showing the lifetimes of the references in the code, and the corresponding constraints.

5.3 Grounded Theory

The works in Sec. 4 use a wide variety of research methods, including surveying existing experience reports, semi-structured interviews, online surveys, and controlled experiments. But so far they have left a notable gap in the methodologies, which is a deep qualitative understanding of how Rust programmers actually write code. Questions such as what tools they use, if and how they read error messages, how they reason about various errors, common debugging strategies, etc. are all left out of the scope of the existing research. To fill this gap, I recommend building a Grounded Theory of how Rust programmers write code in relation to programmers in other languages, or using different type or memory management systems.

Grounded Theory (GT) started in the 1960s as a research method in sociology, but has since become a standard method of qualitative research in many field as a method for developing theories bottom-up through observation [Charmaz and Bryant 2010]. Speaking very broadly, GT as a method is an iterative process of collecting data through interviews and observations and using open-coding to develop a theory that is *grounded* in the empirical data (rather than informed by or confirming existing theories). But the above description is too simplistic. Various versions of GT exist, each of which make different assumptions about the nature of knowledge (positivism vs. constructivism) and the precise steps they follow are subtly different and outside the scope of this report.

That said, GT is an established and popular method in Software Engineering research [Stol et al. 2016], and recently [Lubin and Chasins 2021] employed *Constructivist* GT [Charmaz 2006] to develop a deeper understanding of how statically-typed functional programmers write code. Which is why I recommend building a grounded theory of "How Rust Programmers Write Code". Speculating on what we may learn from this is counter to GT's philosophy. But my hope is that such a GT will help develop the foundations that can both better motivate and help us understand the human-centered study and design choices involved in Rust and similar advanced type systems.

ACKNOWLEDGMENTS

I would like to thank my advisors Sorin Lerner and Nadia Polikarpova for providing feedback on this report, as well as their advice and mentorship. I would also like to thank Hila Peleg for being my de-facto advisor in my first two years in Graduate school, and Ruanqianqian (Lisa) Huang and Shraddha Barke for being amazing collaborators and co-authors whose research has been a major source of motivation and inspiration in my own work.

REFERENCES

Jonathan Aldrich, Valentin Kostadinov, and Craig Chambers. 2002. Alias Annotations for Program Understanding. In *Proceedings of the 17th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Seattle, Washington, USA) *(OOPSLA '02)*. Association for Computing Machinery, New York, NY, USA, 311–330. https://doi.org/10.1145/582419.582448

Titus Barik, Justin Smith, Kevin Lubick, Elisabeth Holmes, Jing Feng, Emerson Murphy-Hill, and Chris Parnin. 2017. Do Developers Read Compiler Error Messages?. In *Proceedings of the 39th International Conference on Software Engineering* (Buenos Aires, Argentina) *(ICSE '17)*. IEEE Press, 575–585. https://doi.org/10.1109/ICSE.2017.59

Brett A. Becker, Paul Denny, Raymond Pettit, Durell Bouchard, Dennis J. Bouvier, Brian Harrington, Amir Kamil, Amey Karkare, Chris McDonald, Peter-Michael Osera, Janice L. Pearce, and James Prather. 2019. Compiler Error Messages Considered Unhelpful: The Landscape of Text-Based Programming Error Message Research. In *Proceedings of the Working Group Reports on Innovation and Technology in Computer Science Education (ITiCSE-WGR '19)*. Association for Computing Machinery, New York, NY, USA, 177–210. https://doi.org/10.1145/3344429.3372508

Jean-Philippe Bernardy, Mathieu Boespflug, Ryan R. Newton, Simon Peyton Jones, and Arnaud Spiwack. 2017. Linear Haskell: practical linearity in a higher-order polymorphic language. *Proceedings of the ACM on Programming Languages* 2, POPL (Dec. 2017), 5:1–5:29. https://doi.org/10.1145/3158093

David Blaser. 2019. Simple Explanation of Complex Lifetime Errors in Rust. https://ethz.ch/content/dam/ethz/special-interest/infk/chair-program-method/pm/documents/Education/Theses/David_Blaser_BA_Report.pdf

Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. 2002. Ownership Types for Safe Programming: Preventing Data Races and Deadlocks. In *Proceedings of the 17th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Seattle, Washington, USA) *(OOPSLA '02)*. Association for Computing Machinery, New York, NY, USA, 211–230. https://doi.org/10.1145/582419.582440

Chandrasekhar Boyapati, Barbara Liskov, and Liuba Shrira. 2003a. Ownership Types for Object Encapsulation. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New Orleans, Louisiana, USA) *(POPL '03)*. Association for Computing Machinery, New York, NY, USA, 213–223. https://doi.org/10.1145/604131.604156

Chandrasekhar Boyapati, Alexandru Salcianu, William Beebee, and Martin Rinard. 2003b. Ownership Types for Safe Region-Based Memory Management in Real-Time Java. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation* (San Diego, California, USA) *(PLDI '03)*. Association for Computing Machinery, New York, NY, USA, 324–337. https://doi.org/10.1145/781131.781168

John Boyland. 2001. Alias burying: Unique variables without destructive reads. *Software: Practice and Experience* 31, 6 (2001), 533–553. https://doi.org/10.1002/spe.370 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.370

Stephen Cass, Preeti Kulkarni, and Erico Guizzo. 2022. Top Programming Languages 2021. https://spectrum.ieee.org/top-programming-languages/

Kathy Charmaz. 2006. *Constructing Grounded Theory: A Practical Guide Through Qualitative Analysis.* SAGE Publishing Inc.

Kathy. Charmaz and Antony Bryant. 2010. Grounded Theory. In *International Encyclopedia of Education (Third Edition)*, Penelope Peterson, Eva Baker, and Barry McGaw (Eds.). Elsevier, Oxford, 406–412. https://doi.org/10.1016/B978-0-08-044894-7.01581-5

Dave Clarke and Tobias Wrigstad. 2003. External Uniqueness Is Unique Enough. In *ECOOP 2003 – Object-Oriented Programming*, Luca Cardelli (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 176–200.

Dave Clarke, Johan Östlund, Ilya Sergey, and Tobias Wrigstad. 2013. Ownership Types: A Survey. In *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*, Dave Clarke, James Noble, and Tobias Wrigstad (Eds.). Springer, Berlin, Heidelberg, 15–58. https://doi.org/10.1007/978-3-642-36946-9_3

David G. Clarke, John M. Potter, and James Noble. 1998. Ownership Types for Flexible Alias Protection. In *Proceedings of the 13th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Vancouver, British Columbia, Canada) *(OOPSLA '98)*. Association for Computing Machinery, New York, NY, USA, 48–64. https://doi.org/10.1145/286936.286947

Michael Coblenz, Jonathan Aldrich, Brad A. Myers, and Joshua Sunshine. 2018. Interdisciplinary Programming Language Design. In *Proceedings of the 2018 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software* (Boston, MA, USA) *(Onward! 2018)*. Association for Computing Machinery, New York, NY, USA, 133–146. https://doi.org/10.1145/3276954.3276965

Michael Coblenz, Michelle Mazurek, and Michael Hicks. 2021. Does the Bronze Garbage Collector Make Rust Easier to Use? A Controlled Experiment. *arXiv preprint arXiv:2110.01098* (2021).

Will Crichton. 2018. What is Systems Programming, Really? https://willcrichton.net/notes/systems-programming/

Will Crichton. 2020. The Usability of Ownership. *arXiv preprint arXiv:2011.06171* (2020).

Will Crichton, Marco Patrignani, Maneesh Agrawala, and Pat Hanrahan. 2021. Modular Information Flow Through Ownership. *CoRR* abs/2111.13662 (2021). arXiv:2111.13662 https://arxiv.org/abs/2111.13662

Matthew Davis, Peter Schachte, Zoltan Somogyi, and Harald Søndergaard. 2012. Towards Region-Based Memory Management for Go. In *Proceedings of the 2012 ACM SIGPLAN Workshop on Memory Systems Performance and Correctness* (Beijing, China) *(MSPC '12)*. Association for Computing Machinery, New York, NY, USA, 58–67. https://doi.org/10.1145/2247684.2247695

Dan Diaper. 2004. Understanding task analysis for human-computer interaction. *The handbook of task analysis for human-computer interaction* (2004), 5–47.

Dietler Dominik. 2018. Visualization of Lifetime Constraints in Rust. https://ethz.ch/content/dam/ethz/special-interest/infk/chair-program-method/pm/documents/Education/Theses/Dominik_Dietler_BA_report.pdf

Ana Nora Evans, Bradford Campbell, and Mary Lou Soffa. 2020. Is Rust Used Safely by Software Developers?. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 246–257. https://doi.org/10.1145/3377811.3380413

Manuel Fahndrich and Robert DeLine. 2002. Adoption and Focus: Practical Linear Types for Imperative Programming. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation* (Berlin, Germany) *(PLDI '02)*. Association for Computing Machinery, New York, NY, USA, 13–24. https://doi.org/10.1145/512529.512532

Kelsey R Fulton, Anna Chan, Daniel Votipka, Michael Hicks, and Michelle L Mazurek. 2021. Benefits and drawbacks of adopting a secure programming language: rust as a case study. In *Seventeenth Symposium on Usable Privacy and Security (SOUPS 2021)*. 597–616.

Gongming, Luo, Vishnu Reddy, Marcelo Almeida, Yingying Zhu, Ke Du, and Cyrus Omar. 2020. RustViz: Interactively Visualizing Ownership and Borrowing. *arXiv:2011.09012 [cs]* (Nov. 2020). http://arxiv.org/abs/2011.09012 arXiv: 2011.09012.

Dan Grossman, Michael Hicks, Trevor Jim, and Greg Morrisett. 2005. Cyclone: A type-safe dialect of C. *C/C++ Users Journal* 23, 1 (2005), 112–139.

Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. 2002a. Region-Based Memory Management in Cyclone. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation* (Berlin, Germany) *(PLDI '02)*. Association for Computing Machinery, New York, NY, USA, 282–293. https://doi.org/10.1145/512529.512563

Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. 2002b. Region-based memory management in cyclone. *ACM SIGPLAN Notices* 37, 5 (May 2002), 282–293. https://doi.org/10.1145/543552.512563

Philipp Haller and Martin Odersky. 2010. Capabilities for Uniqueness and Borrowing. In *ECOOP 2010 – Object-Oriented Programming*, Theo D'Hondt (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 354–378.

Eric Holk, Ryan Newton, Jeremy Siek, and Andrew Lumsdaine. 2014. Region-Based Memory Management for GPU Programming Languages: Enabling Rich Data Structures on a Spartan Host. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications* (Portland, Oregon, USA) *(OOPSLA '14)*. Association for Computing Machinery, New York, NY, USA, 141–155. https://doi.org/10.1145/2660193.2660244

Christopher D. Hundhausen, Sarah A. Douglas, and John T. Stasko. 2002. A Meta-Study of Algorithm Visualization Effectiveness. *Journal of Visual Languages & Computing* 13, 3 (2002), 259–290. https://doi.org/10.1006/jvlc.2002.0237

Herman Ruge Jervell. 1996. Jean-Yves Girard. Linear logic. Theoretical computer science, vol. 50 (1987), pp. 1–101. - A. S. Troelstra. Lectures on linear logic. CSLI lecture notes, no. 29. Center for the Study of Language and Information, Stanford 1992, also distributed by Cambridge University Press, New York, ix 200 pp. *Journal of Symbolic Logic* 61, 1 (1996), 336–338. https://doi.org/10.2307/2275616

Trevor Jim, J. Morrisett, Dan Grossman, Michael Hicks, James Cheney, and Yanling Wang. 2002. Cyclone: A safe dialect of C. *Proc. of the 2002 USENIX Annual Technical Conference*, 275–288.

Steve Klabnik and Carol Nichols. 2017. *The Rust Programming Language*. No Starch Press, San Francisco, CA, USA.

Justin Lubin and Sarah E. Chasins. 2021. How statically-typed functional programmers write code. *Proceedings of the ACM on Programming Languages* 5, OOPSLA (Oct. 2021), 155:1–155:30. https://doi.org/10.1145/3485532

Henning Makholm. 2000. *Region-Based Memory Management in Prolog*. Technical Report. In Proceedings of the 2nd International Symposium on Memory Management.

Nicholas D. Matsakis and Felix S. Klock. 2014. The Rust Language. In *Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology* (Portland, Oregon, USA) *(HILT '14)*. Association for Computing Machinery, New York, NY, USA, 103–104. https://doi.org/10.1145/2663171.2663188

Joseph Mcgrath, E. 1995. Methodology Matters: Doing Research in the Behavioral and Social Sciences. In *Readings in Human–Computer Interaction*, Ronald Baecker, M., Jonathan Grudin, William Buxton, A.S., and Saul Greenberg (Eds.). Morgan Kaufmann, 152–169. https://doi.org/10.1016/B978-0-08-051574-8.50019-4

Federico Mena-Quintero. 2018. Rust things I miss in C. https://web.archive.org/web/20210119090858/https://people.gnome.org/~federico/blog/rust-things-i-miss-in-c.html

Greg Morrisett, Karl Crary, Neal Glew, Dan Grossman, Richard Samuels, Frederick Smith, David Walker, Stephanie Weirich, and Steve Zdancewic. 1999. TALx86: A Realistic Typed Assembly Language. In *Second ACM SIGPLAN Workshop on*

Compiler Support of System Software. Atlanta, GA, USA, 25–35.

Khanh Nguyen, Lu Fang, Guoqing Xu, and Brian Demsky. 2015. Speculative Region-Based Memory Management for Big Data Systems. In *Proceedings of the 8th Workshop on Programming Languages and Operating Systems* (Monterey, California) *(PLOS '15)*. Association for Computing Machinery, New York, NY, USA, 27–32. https://doi.org/10.1145/2818302.2818308

Benjamin C. Pierce. 2004. *Advanced Topics in Types and Programming Languages*. The MIT Press.

James Prather, Raymond Pettit, Kayla Holcomb McMurry, Alani Peters, John Homer, Nevan Simone, and Maxine Cohen. 2017. On Novices' Interaction with Compiler Error Messages: A Human Factors Approach. In *Proceedings of the 2017 ACM Conference on International Computing Education Research* (Tacoma, Washington, USA) *(ICER '17)*. Association for Computing Machinery, New York, NY, USA, 74–82. https://doi.org/10.1145/3105726.3106169

Boqin Qin, Yilun Chen, Zeming Yu, Linhai Song, and Yiying Zhang. 2020. 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* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 763–779. https://doi.org/10.1145/3385412.3386036

The Rust Project Developers. 2022. *The Rustonomicon*. https://doc.rust-lang.org/nomicon/

Nischal Shrestha, Colton Botta, Titus Barik, and Chris Parnin. 2020. Here We Go Again: Why is It Difficult for Developers to Learn Another Programming Language?. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 691–701. https://doi.org/10.1145/3377811.3380352

Juha Sorva, Ville Karavirta, and Lauri Malmi. 2013. A Review of Generic Program Visualization Systems for Introductory Programming Education. *ACM Transactions on Computing Education* 13, 4 (Nov. 2013), 1–64. https://doi.org/10.1145/2490822

Stack Overflow. 2016. Stack Overflow Developer Survey 2016 Results. https://insights.stackoverflow.com/survey/2016

Stack Overflow. 2017. Stack Overflow Developer Survey 2017. https://insights.stackoverflow.com/survey/2017/?utm_source=so-owned&utm_medium=social&utm_campaign=dev-survey-2017&utm_content=social-share

Stack Overflow. 2018. Stack Overflow Developer Survey 2018. https://insights.stackoverflow.com/survey/2018/?utm_source=so-owned&utm_medium=social&utm_campaign=dev-survey-2018&utm_content=social-share

Stack Overflow. 2019. Stack Overflow Developer Survey 2019. https://insights.stackoverflow.com/survey/2019/?utm_source=social-share&utm_medium=social&utm_campaign=dev-survey-2019

Stack Overflow. 2020. Stack Overflow Developer Survey 2020. https://insights.stackoverflow.com/survey/2020/?utm_source=social-share&utm_medium=social&utm_campaign=dev-survey-2020

Stack Overflow. 2021. Stack Overflow Developer Survey 2021. https://insights.stackoverflow.com/survey/2021/?utm_source=social-share&utm_medium=social&utm_campaign=dev-survey-2021

Klaas-Jan Stol, Paul Ralph, and Brian Fitzgerald. 2016. Grounded theory in software engineering research: a critical review and guidelines. In *Proceedings of the 38th International Conference on Software Engineering (ICSE '16)*. Association for Computing Machinery, New York, NY, USA, 120–131. https://doi.org/10.1145/2884781.2884833

The Rust Core Team. 2018. Announcing Rust 1.31 and Rust 2018. https://blog.rust-lang.org/2018/12/06/Rust-1.31-and-rust-2018.html

The Rust Survey Team. 2020. Rust Survey 2020 Results. https://blog.rust-lang.org/2020/12/16/rust-survey-2020.html

TIOBE Software BV. 2022. index. https://www.tiobe.com/tiobe-index/

Mads Tofte and Jean-Pierre Talpin. 1997. Region-based memory management. *Information and computation* 132, 2 (1997), 109–176.

Philip Wadler. 1990. Linear Types can Change the World!. In *Programming Concepts and Methods*.

David Walker and Kevin Watkins. 2001. On Regions and Linear Types (Extended Abstract). In *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming* (Florence, Italy) *(ICFP '01)*. Association for Computing Machinery, New York, NY, USA, 181–192. https://doi.org/10.1145/507635.507658

Aaron Weiss, Olek Gierczak, Daniel Patterson, and Amal Ahmed. 2021. Oxide: The Essence of Rust. *arXiv:1903.00982 [cs]* (Oct. 2021). http://arxiv.org/abs/1903.00982 arXiv: 1903.00982.

Hongwei Xi and Frank Pfenning. 1999. Dependent Types in Practical Programming. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Antonio, Texas, USA) *(POPL '99)*. Association for Computing Machinery, New York, NY, USA, 214–227. https://doi.org/10.1145/292540.292560

Anna Zeng and Will Crichton. 2019. Identifying barriers to adoption for Rust through online discourse. *arXiv preprint arXiv:1901.01001* (2019).

Shuofei Zhu, Ziyi Zhang, Boqin Qin, Aiping Xiong, and Linhai Song. 2022. Learning and Programming Challenges of Rust: A Mixed-Methods Study. (2022), 13. https://doi.org/10.1145/3510003.3510164