Master Thesis

Julius-Maximilians-
UNIVERSITÄT
WÜRZBURG

Visualizing Ownership and Borrowing in Rust Programs

Schott Christian
Department of Computer Science
Chair of Computer Science II (Software Engineering)

Prof. Dr.-Ing. Samuel Kounev
First Reviewer

Lukas Beierlieb M.Sc.
First Advisor

Submission
07. May 2024

www.uni-wuerzburg.de

Abstract

While Rust is popular among developers due to its performance and memory safety features, its widespread adoption has remained relatively slow. This can be partially attributed to the perceived difficulty associated with learning its unique features, particularly the borrow checker.

This thesis explores the viability of using visualization techniques to demystify Rust's ownership-based memory management system, with the aim of lowering the steep learning curve associated with the language. Previous research in this area indicates that such visualizations can assist the teaching of Rust's particular memory management concepts. Insights gleaned from the evaluation of these existing visualization tools have informed the development of a novel approach for visualizing Rust's memory management, addressing some limitations of existing works.

The proposed approach has been implemented as a standalone tool, capable of automatically generating visualizations for user-provided code. This tool leverages a user-friendly interface and intuitive graphics to visualize Rust's complex memory management concepts, making them more accessible to developers.

The effectiveness of this tool is assessed through a comparative analysis with previous works. Additionally, a qualitative survey involving developers with varying levels of Rust proficiency is conducted to evaluate the tool's impact on their understanding of Rust's ownership and borrowing model.

Preliminary results suggest that the use of visualizations can enhance comprehension and facilitate the understanding of Rust's ownership and borrowing model, particularly among novice Rust developers. The findings indicate that such a tool could potentially lower the learning curve associated with Rust, thereby accelerating its adoption rate.

Zusammenfassung

Trotz der Beliebtheit der Rust Programmiersprache unter Entwicklern aufgrund ihrer Leistungsfähigkeit und Speichersicherheitsfunktionen ist ihre Verbreitung in der Softwareentwicklungsbranche noch begrenzt. Ein Grund dafür ist die wahrgenommene Komplexität der Sprache, insbesondere im Zusammenhang mit dem sogenannten "Borrow Checker".

In dieser Arbeit wird der potenzielle Nutzen von Visualisierungen der auf "Ownership" basierenden Speicherverwaltung von Rust untersucht, um die steile Lernkurve, die mit der Sprache verbunden ist, zu senken. Frühere Studien in diesem Bereich haben bereits einen positiven Effekt von Visualisierungen beim Lehren der Besonderheiten der Speicherverwaltung von Rust festgestellt. Erkenntnisse aus der Untersuchung dieser bestehenden Visualisierungswerkzeuge haben zur Entwicklung eines neuen Ansatzes für die Visualisierung beigetragen, der einige der bisherigen Einschränkungen beseitigt.

Der vorgeschlagene Ansatz wurde als eigenständiges Programm implementiert, das in der Lage ist, automatisch Visualisierungen für vom Benutzer bereitgestellten Code zu erzeugen. Dieses Werkzeug bietet eine benutzerfreundliche Schnittstelle zur Erstellung intuitiver Grafiken, die die komplexen Konzepte der Speicherverwaltung von Rust veranschaulichen und für Entwickler leichter zugänglich machen.

Die Wirksamkeit dieses Werkzeugs wird durch einen Vergleich mit früheren Arbeiten bewertet. Zusätzlich wurde eine qualitative Umfrage unter Entwicklern mit unterschiedlichem Kenntnisstand in Rust durchgeführt, um die Auswirkungen des Visualisierungsprogramms auf ihr Verständnis des "Ownership"- und "Borrowing"-Modells von Rust zu bewerten.

Vorläufige Ergebnisse deuten darauf hin, dass die Verwendung von Visualisierungen das Verständnis des Speichermanagementkonzepts von Rust erleichtern kann, insbesondere bei Entwicklern mit wenig Erfahrung in der Sprache. Diese Ergebnisse legen nahe, dass ein solches Werkzeug die steile Lernkurve, die mit Rust verbunden ist, verringern und somit seine Verbreitung beschleunigen könnte.

Contents

1 Introduction

The selection of a programming language for an application is often dictated by the performance criticality of the task at hand. For applications where performance is less of a concern, higher-level languages such as JavaScript or C# are typically favored due to their automatic memory management and user-friendly nature. This preference is also reflected in the Stack Overflow Developer Survey 2023 [1], which reveals that eight out of the top ten most utilized general-purpose programming languages are such higher-level languages, excluding markup and shell languages like HTML and Bash. Conversely, low-level systems programming languages like C and C++, only accounted for two of the top ten. While these languages offer full access to system resources, they also demand meticulous attention from the developer to ensure memory safety.

Such low-level languages, while capable of producing highly performant software, are also a common source of memory-related bugs. These bugs represent a significant proportion of software vulnerabilities that can be exploited by third parties [2]. In a recent blog post from the Microsoft Security Response Centre, they disclosed that around 70% of security patches in their products were related to memory safety issues [3].

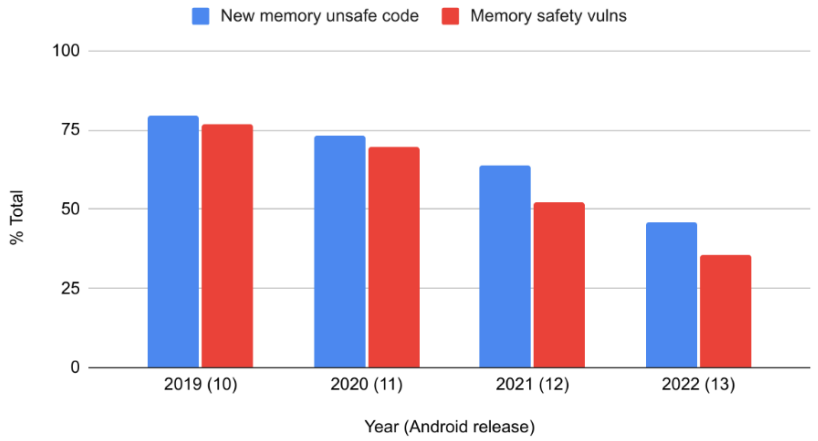


Figure 1.1: Drop in the proportion of memory-related safety vulnerabilities in Android, correlated to the proportion of new code written in non memory safe languages [4].

Similarly, around 76% of vulnerabilities in the Android operating system were related to

memory safety in 2019. However, since then security researchers at Google have observed a drop in the proportion of memory safety related vulnerabilities to only 35% in 2022 [4]. This drop in memory-related vulnerabilities correlates strongly with their focus on utilizing memory safe languages, like Rust, for new code introduced to Android, as shown in Figure 1.1.

Rust [5], a modern system-level programming language, allows developing programs with performance on par with C or C++, while ensuring memory safety. This is achieved through a mechanism known as 'ownership', which is the main focus of this thesis. Although Rust did not invent the concept of ownership [6], it is the first language to combine it with beneficial design elements from other languages into a unified framework. This includes, for example, algebraic data types and pattern matching from functional languages, references and move semantics from C++, and closures from Ruby [7].

The aggregation of these features has caused Rust to be voted as the most admired programming language in the annual Stack Overflow Developer Survey. Meaning that developers who chose to learn Rust also wanted to continue using the language in subsequent years [8]. It has held this position in the survey continuously ever since its first stable release in 2015 [9].

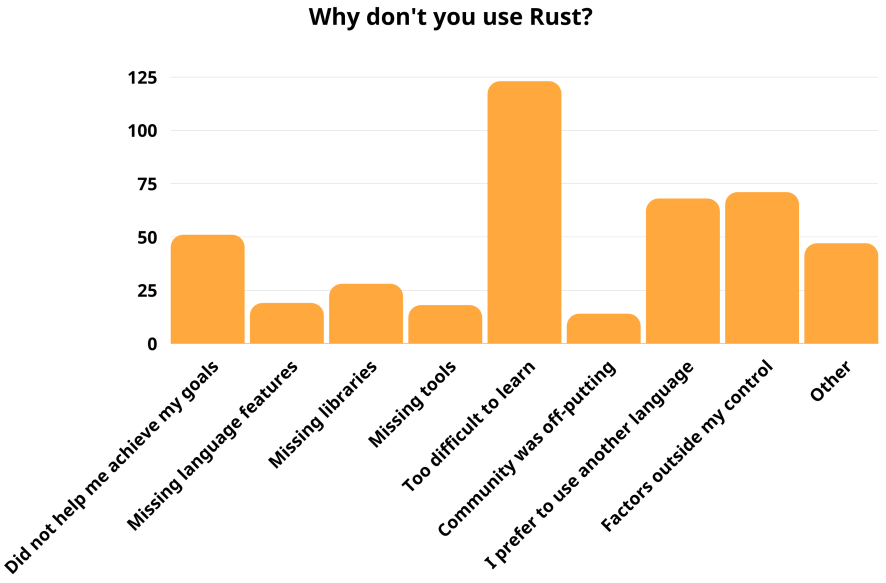


Figure 1.2: Reasons non-Rust developers ($\sim$ 10% of the 9433 total participants) cited for not learning Rust in the official 2022 Rust survey [10].

However, this raises the intriguing question of why Rust is not used more widely in the industry, given its popularity among developers. One significant factor is certainly the difficulty for long-standing projects, initiated before Rust's introduction, to transition to a new language. Consequently, many of these projects will continue to rely on languages like C and C++ for the foreseeable future. Nevertheless, even very large scale projects, like the Linux and Windows operating systems, have recently laid the ground work for adopting Rust in their development [11] [12].

Another contributing factor is that many of Rust's distinctive design elements are unfamiliar to programmers coming from other higher-level languages, making the initial learning process more challenging [13]. In the 2022 annual Rust survey conducted by the Rust Foundation, approximately 26% of non-Rust users cited its perceived difficulty as the primary deterrent for learning it (see Figure 1.2), second only to time constraints, cited by around 62% of non-Rust users.

In particular, Rust's borrow checker, the main topic of discussion in Section 2.3, necessitates a shift in coding approach, which has been identified as one of the most significant challenges when adopting Rust [14] [15]. This was also reflected in the 2020 Rust survey, where 'ownership' and 'lifetimes' were rated as the top two most difficult topics (see Figure 1.3).

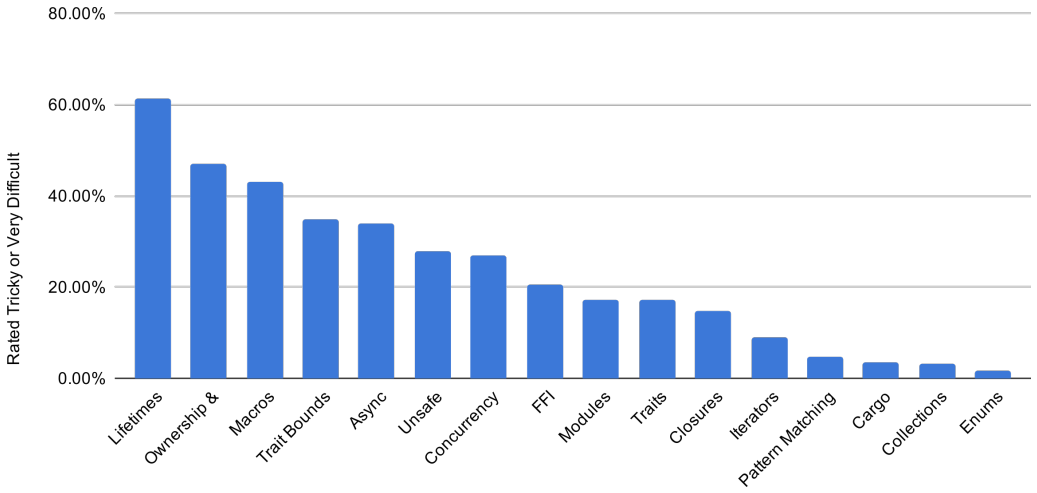


Figure 1.3: Topics rated as tricky or very difficult by participants of the 2020 official Rust survey [16].

Rust's image of being a complex language with a steep learning curve has resulted in a gradual, yet consistent, increase in its adoption by developers [13]. W. Crichton [17] suggests that the development of tools to visualize Rust's memory management could facilitate the construction of a mental model of the borrow checker, thereby accelerating the learning process. By providing a visual representation of ownership and lifetimes, these tools could potentially reduce the cognitive load associated with reading and writing Rust code. This would allow developers to concentrate more effectively on the fundamental logic of the program at hand, instead of particularities of the language.

The next Chapter 2 provides explanations for the most important terminology used in the rest of this thesis. In the following Chapter 3 existing tools for generating such visualizations are presented, and evaluated. Based on this evaluation of previous work, a novel visualization approach is described and implemented in Chapters 4 and 5. These visualizations are then evaluated on multiple code examples and compared to previous works in Chapter 6. Additionally, a survey has been conducted for evaluating the usability of the resulting tool for programmers of different familiarity levels with the Rust language qualitatively. Finally, Chapter 7 summarizes the thesis' findings and proposes possible directions for future work.

2 Background

Before going into detail about generating visualizations of Rust's memory management model, the following Section 2.1 provides a basic overview over memory management in general, defining the most important vocabulary needed for the following chapters. Section 2.3 then introduces and explains the main concepts behind Rust's ownership based memory management, giving a short introduction to Rust's notorious borrow checker.

2.1 Memory Management

Memory management is a complex topic with many interlocking systems, but for this thesis a rather high-level understanding is sufficient.

At the physical level, a computer stores information in binary format, where each single value is either low (0) or high (1), termed as a bit. Given that a single bit is typically not useful on its own, the smallest unit of memory usually referred to is one byte, which is a collection of eight bits. A sequence of such bytes can encode diverse types of information, including text, images, or programs. A computer program, essentially a sequence of encoded instructions, can be executed by the Central Processing Unit (CPU) of the computer.

For running a program, the Operating System (OS) loads the instruction sequence from a long-term storage device like a hard drive into the computer's main or working memory, known as Random Access Memory (RAM). To each running program, or process, it appears as if it has its own separate memory space available to it. A memory space being a consecutive sequence of bytes, where each byte can be uniquely identified by a memory address. The OS is responsible for mapping the virtual process memory addresses to actual memory addresses of the underlying hardware, but these details are not relevant for this thesis.

All C-based languages, which all languages this thesis is referring to are, divide their memory into four fundamental segments [18]:

- Code or Text segment: instruction sequence of the program
- Data segment: global, static, constant, and uninitialized data
- Heap: memory allocated dynamically during the runtime of the program
- Stack: local variables, function arguments, and return values

For the purposes of this thesis, the heap and stack segments are the two most relevant segments, and thus discussed further in the following sections.

2.1.1 Stack

As the name suggests, memory allocated on the stack is placed on top of what is there already, and whatever is on top can also be taken back off, adhering to the first in, first out principle. This results in the allocation and deallocation of stack memory being very simple and fast, as the stack grows and shrinks linearly.

The stack is the location where the memory of all local variables, function arguments, and return values resides. Whenever a function is invoked, a new stack frame, containing the memory of the invoked function, is pushed onto the stack. Upon the function's return, the stack frame is removed from the top of the stack, thereby deallocating all memory owned by the function. This implies that a function can only access its own memory, the memory of stack frames beneath it, and values returned from invoked functions.

For memory to persist beyond the scope of a function, it must be allocated on the heap. Moreover, the size of memory allocated on the stack must be known at the time of compilation, requiring allocation of dynamically sized memory blocks to be deferred to the heap.

2.1.2 Heap

The heap is a range of memory, independent of function calls, where allocations of unknown size at compile time or long-lived allocations can be stored. Memory on the heap has to be allocated and deallocated by the program manually, and failing to do so properly can lead to common memory-related errors described in Section 2.1.3.

When compared to stack memory, the process of allocating or deallocating a chunk of memory on the heap is considerably more complex than simply pushing or popping from the end. Therefore, the standard libraries of most low-level languages provide higher-level abstractions that handle most of the internal details of partitioning heap memory or requesting additional memory from the OS. Examples of these abstractions include `malloc`/`free` in C, and `new`/`delete` in C++.

2.1.3 Memory Errors

In languages with manual memory management, such as C or C++, specific memory addresses of the process can be read or written to directly. This direct access enables the creation of very fast and efficient programs, but it also introduces a lot of complexity and potential for memory-related bugs. For instance, reading from an incorrect memory address could result in invalid data being read. Similarly, writing to an incorrect address could unintentionally overwrite other data, leading to unintended behavior or crashes later.

With heap-allocated memory the responsibility for ensuring that the memory is freed again, and that freed memory is not accessed afterward, lies with the developer. Failure to maintain this invariant can lead to program crashes or unexpected behavior.

Common examples of memory-related errors include:

- Memory Leak: Memory that has been allocated is never freed again, causing the memory consumption of a program to grow indefinitely.
- Use After Free: Once a section of memory is freed again, it may be reused by another allocation. In this case, unintended data may be read or overwritten when accessing the freed memory address.

- Buffer Overflow: When more data is written to a memory location than was allocated for it, neighboring data may be overwritten. This is especially problematic when the written data originates from the user, as it allows a potential attacker to overwrite the program's data, altering its behavior [19].

2.1.4 Garbage Collection

Programming languages such as JavaScript or C# abstract away dynamic memory allocations on the heap behind a so-called Garbage Collector (GC), leading to these languages often being referred to as managed languages. Instead of requiring developers freeing heap memory manually, managed languages track allocated heap memory and automatically free it if it is no longer in use. This tracking and freeing is performed by a separate thread, the GC, which runs at set intervals and checks which allocations are no longer needed. This prevents memory leaks, as developers cannot forget to free allocated resources. However, the GC must interrupt the normal execution of the program, leading to generally degraded performance and more unpredictable response times.

Additionally, managed languages typically disallow direct access to memory addresses, completely eliminating a whole category of memory-related bugs described in Section 2.1.3. However, this also prevents some optimizations that are possible in languages with direct memory access.

For instance, a managed language like Java must perform a bound check when accessing a specific index of an array to ensure the safety of the access, explicitly throwing an exception otherwise [20]. In contrast, in a language with direct memory access, the contents of the given index can be read directly, without any additional checks, making the access slightly faster. However, the responsibility of ensuring the safety of the access then lies with the developer.

2.2 Memory Aliasing

In all C-based languages, a mechanism for referring to a memory location exists, typically known as a pointer, as it points to a specific memory address. Pointers are primarily used to store the address of heap-allocated memory on the stack, but generally they can be used for pointing to any memory address. Managed languages also utilize pointers for internal access to heap-allocated objects, but they largely conceal the lower-level memory access from the developer.

Memory aliasing is a phenomenon that arises when a memory region, or a portion thereof, is referenced from multiple locations in the source code. This means that several pointers can point to the same memory location, with the potential to read from or write to the same memory. While aliasing in itself is not problematic, it frequently leads to bugs, as writing to one pointer may inadvertently alter the value read from another. This becomes particularly problematic in multi-threaded code, where multiple threads may access the same memory, leading to race conditions. The detection and resolution of such race conditions is a complex task and is a subject of ongoing research [21].

2.3 Rust's Memory Management

Rust aims to prevent previously mentioned memory-related errors and aliasing by enforcing strict rules on memory access. These rules ensure that any code containing such issues fails to compile, thereby necessitating the creation of robust and secure programs by developers. Consequently, Rust programs maintain a well-defined state during execution, leaving less exploitable weaknesses and unexpected behavior. This generally leads to the creation of safer and more stable applications.

2.3.1 Ownership

Ownership is the core mechanism behind Rust's memory management approach, eliminating the need for a GC. This concept is closely tied to the Resource Acquisition Is Initialization (RAII) pattern and move semantics, both introduced by C++. The key distinction lies in Rust's enforcement of these patterns at the language level, as opposed to being design patterns in C++ [22].

The RAII pattern essentially binds the lifecycle of a resource (such as memory allocated on the heap) to the lifecycle of a variable. This binding implies that the linked resource remains valid as long as the variable is accessible. Once the variable goes out of scope, the linked resource can be freed, or 'dropped' in Rust's terminology. As the language ensures resource freeing, developers cannot overlook this step, thereby preventing memory leaks.

In Rust, the variable to which a resource is assigned is termed the *owner* of that resource. For instance, in the first line of Listing 2.1, the `String` value `"Hello World"` is assigned to the owner `x`.

```
1 let x = String::from("Hello␣World!"); // x is the owner of the string
2
3 let y = x;       // ownership of the string is moved to y
4
5 print!("{}", x); // compile error! x was moved in line 3
```

Listing 2.1: Basic example of ownership in Rust.

Line 3 of Listing 2.1 demonstrates another aspect of Rust's ownership model. When a variable's value is assigned to another variable, the ownership of the value is *moved*, leaving the original variable in an uninitialized state. Any attempt to access a variable's value after it has been moved is prohibited by the compiler.

Rust's ownership rules can be summarized as follows:

- Each value in Rust has an **owner**.
- Only **one** owner can exist at any given time.
- When the **owner** goes out of scope, the value is **dropped** and its memory is freed.

One special case to these ownership rules are primitive types such as numbers. When these primitive values are used, they are implicitly copied rather than moved, leaving the owning variable in a valid and unchanged state.

In Rust, a primitive type is defined as any type that implements the language-defined `Copy` trait. The trait system in Rust is a separate topic and is not within the scope of this thesis, hence further explanation is not provided here. Comprehensive information about ownership and traits can be found in the official Rust book [23].

```
1 let x = String::new();
2 x.push_str("Hello");  // error! x may not be mutated
3
4 let mut x = x;        // redefinition of x, marked as mutable
5 x.push_str("Hello");  // works!
```

Listing 2.2: Example of mutability in Rust.

Another important difference of Rust, compared to many other C-based languages, is that all variables are read-only or immutable by default. In order to modify a value owned by

a variable, the variable has to be annotated as being mutable explicitly. This is achieved by annotating the variable with the `mut` keyword, as demonstrated in Listing 2.2.

Line 4 of Listing 2.2 is known as a redefinition of `x`, as the value inside `x` is *moved* to a new owner also named `x`. Any subsequent usage of `x` refers to its new definition, which is marked as mutable in this case.

2.3.2 Borrowing

Relying solely on ownership would render the creation of efficient programs with Rust unfeasible, as moving values also entails copying the underlying memory in most cases. Thus, Rust incorporates a second mechanism for memory management, known as borrowing.

Instead of taking ownership to a value, e.g., when passing arguments to a function, a *reference* to the value can be created using the `&`-operator, permitting the value to be read indirectly. Conversely, the `&mut`-operator can be employed to create a *mutable reference* to a mutable variable, thereby enabling the value to be written to.

For instance, the `push_str` method from Listing 2.2 in the previous section has a method signature of:

```
1 pub fn push_str(&mut self, string: &str) { ... }
```

The `self`-parameter in Rust is similar to the `this`-parameter in C++. However, in Rust, method signatures must define this `self`-parameter explicitly. The special `&mut self` syntax is a short form, for a `self`-parameter of type `&mut Self`, where the `Self`-type refers to `String` in this case. This method takes a mutable reference to the value of the `String` as the first parameter and an immutable reference to the appended text as the second parameter.

Fundamentally, a *reference* in Rust is a pointer with some additional constraints:

- References must always be valid, implying that the targeted value must not be moved, dropped, or altered in any way while it is being referred to.
- At any given moment, there can be **either one mutable** reference **or any number of immutable** references to a value.

The first rule ensures that the memory region to which the reference points is always in a valid state, making reading from or writing to the reference well-defined.

The second rule mitigates aliasing issues, as previously discussed in Section 2.2. When holding an *immutable* reference to a value, it can be assumed that the referenced value remains unchanged, as the existence of an additional *mutable* reference would violate the borrowing rules.

Consequently, mutable references are occasionally referred to as *unique* references, as they provide exclusive access to the underlying resource. In contrast, immutable references can be viewed as *shared* references, as multiple immutable references can refer to the same resource.

Similar to other values, references are also assigned to an owner, adhering the RAII pattern. As Rust prohibits accessing a variable before its initialization, it is impossible to access a variable containing an uninitialized reference. In other low-level languages, such as C++, and even managed languages like Java, uninitialized pointers are often set to a `NULL` value, indicating their invalidity [24]. However, without any guarantees regarding whether a pointer may or may not be `NULL` at any given time, each pointer dereference could potentially access unintended memory.

In contrast, Rust requires explicitly encoding potentially uninitialized references in its type system using a `Option`- or `Result`-type, akin to other functional programming languages [25]. This forces developers to explicitly handle cases of potentially uninitialized references.

2.3.3 Lifetimes

The borrowing rules, as outlined in the previous Section 2.3.2, are stated in terms of the 'time' for which a reference is valid, also known as the *lifetime* of the reference. Prior to Rust version 1.63, this lifetime was determined based on scope, similar to the lifetime of local variables, as demonstrated in Listing 2.3.

```
let mut data = vec!['a', 'b', 'c'];
{
    let slice = &mut data[..]; // <-+ 'lifetime
    capitalize(slice);         //   |
} // <------------------------------+
data.push('d');                // OK
```

Listing 2.3: Example of scope-based lifetimes pre Rust 1.63.

However, this approach was found to be overly restrictive in certain scenarios. With the introduction of Rust version 1.63, a new algorithm for determining the lifetime of references was stabilized, known as Non-Lexical Lifetimes (NLL). This novel algorithm considers a reference to be 'alive' from its initialization to its final usage within the Control Flow Graph (CFG) of the program. Listing 2.4 presents an example that compiles with NLL, but would have been deemed unsound by the borrow checker prior to Rust version 1.63, as the reference stored in `slice` would have been considered life until the end of the scope, aliasing with the `data.push('d')` call in line 4. More detailed information and examples can be found in the initial proposal of NLL [26].

```
let mut data = vec!['a', 'b', 'c'];
let slice = &mut data[..];
capitalize(slice);         // last usage of slice
data.push('d');            // OK (with NLL)
```

Listing 2.4: Example of Non-Lexical Lifetimes (NLL).

2.3.3.1 Lifetime Annotations

Similar to other values in Rust, references can be transferred between owners. More specifically, immutable references implement the `Copy`-trait and are thus copied instead of moved. In contrast, mutable references cannot be copied because doing so would violate the uniqueness rule of mutable references.

This process of moving and copying references, however, presents a significant challenge when determining the lifetime of a reference.

```
let mut x = 0;
let ref_x = &x;
let ref2_x = ref_x;
x = 42;                 // compile error! x is still borrowed
println!("{}", ref2_x); // reference to x is used here
```

Listing 2.5: Example of indirect lifetime extension of a reference.

In the example given in Listing 2.5, the assignment to `x` in line 4 would be disallowed, as `x` is still considered borrowed. Despite the fact that the original reference `ref_x` is not used after the assignment, the reference remains 'alive' because it was copied to `ref2_x`, which is used in line 5. In such straightforward cases, the compiler can automatically infer that `ref2_x` also references `x`.

```
1 fn longest(x: &str, y: &str) -> &str {
2     if x.len() > y.len() {
3         x
4     } else {
5         y
6     }
7 }
```

Listing 2.6: Example of a function, returning a reference to the longer input text [27].

However, in a more complex example, as shown in Listing 2.6, the compiler is unable to automatically infer how the returned reference relates to the input references. In this scenario, the function's signature must be manually annotated to indicate to the compiler how the references relate to each other. The correct signature for Listing 2.6 would appear as follows:

```
1 fn longest<'a>(x: &'a str, y: &'a str) -> &'a str
```

This informs the compiler that both references passed as parameters to this function must remain valid for at least as long as the returned reference. In some basic cases, where only one way of relating a function's references is possible, no explicit annotations are needed. This is referred to as lifetime elision [27].

`Struct` and `enum` types, which contain references as parts of their fields, also require such explicit lifetime annotations, enabling the compiler to properly track references concealed within those data types [27].

The precise syntax and semantics of such lifetime annotations are not crucial for the purposes of this thesis. For the analysis proposed in this thesis, it is only essential that the compiler is capable of tracking the relationships between references throughout the program. More detailed information about such lifetime annotations can be found in the official Rust book [27].

2.3.4 Borrow Checker

The borrow checker constitutes a component of the Rust compiler. Its role is to validate all ownership and borrowing rules within a given program prior to the generation of a final executable file. If any issues are detected, the compilation process is aborted, and the developer is provided with an error message explaining why the submitted source code is unsound.

These borrow checker errors can be frustrating for developers, as certain programming patterns, which may induce undefined behavior under certain circumstances but would be accepted by other compilers, are rejected in Rust. The objective of this thesis is to render the ownership and borrowing rules more intuitive for developers, thereby facilitating the process of understanding the borrow checker, and rectifying errors detected by it.

2.4 `unsafe`-Rust

When a Rust program passes the borrow checker's validation, it is generally assured to be memory safe. However, there exists a subset of programs that are indeed memory safe,

but their safety cannot be proven by the borrow checker. In such ambiguous instances, the borrow checker remains conservative and fails to compile the program to ensure safety.

In the realm of `unsafe`-Rust, certain operations, previously disallowed, become permissible. These include dereferencing a raw pointer or invoking an `unsafe`-function, thereby enabling the handling of such edge cases. Access to the `unsafe` components of Rust is granted solely on an opt-in basis, either by marking a function with the `unsafe`-keyword or within an `unsafe`-block. Opting into `unsafe`-Rust shifts the responsibility of maintaining Rust's safety guarantees onto the developer. This necessitates a profound understanding of Rust's internal mechanisms and should be avoided when possible [28]. The topic of `unsafe`-Rust is extensive and complex, and is considered beyond the scope of this thesis.

In Rust, developers are generally advised to use the `unsafe`-feature sparingly. These usages should be self-contained and encapsulated within safe abstractions. An analysis conducted in 2020 on 31,867 public Rust packages revealed that 76.4% of them did not directly use any `unsafe`-Rust features. This leaves 23.6% of packages that do use some `unsafe`-code. However, most of these usages are short and well encapsulated, separating them from the rest of the program [29]. Given that the majority of Rust projects do not require `unsafe`-features, this thesis, which is primarily aimed at novice Rust developers, does not focus on `unsafe`-Rust.

3 Related Work

The concept of visually representing the internal analysis performed by the Rust compiler is not novel. Multiple studies have investigated methods for visualizing aspects of the Rust language, with most of them also focusing on Rust's memory management. These visualizations usually serve two main purposes: they facilitate the learning process for developers new to Rust or provide developmental aid.

3.1 Aquascope

Aquascope is part of an effort by the Cognitive Engineering Lab [31], which focuses on creating human-centered software systems, supporting developers with complex cognitive tasks. Including the development of tools for improving the usability of Rust.

The team began by identifying frequently reported issues with understanding Rust's ownership on StackOverflow. From this, they extracted a set of questions, known as the Ownership Inventory [30], to quantitatively evaluate developers' comprehension of these concepts.

A study of responses from 36 Rust learners to the Ownership Inventory questions revealed that while many could identify the surface-level problems posed by the questions, they often lacked a deeper understanding of *why* a given behavior would be problematic. This led to the development of a tool capable of annotating source code sections with ownership information and generating visualizations of the program's internal memory layout. The aim is to intuitively demonstrate the potential problems that Rust's borrow checker prevents by showing the effects of a given source code line on the program's memory during execution.

Figure 3.1 presents an example program annotated by *Aquascope*, illustrating how modifying a value `v` (`L2`) may invalidate a reference `n` to memory owned by `v`. In this case, the vector `v` may have to reallocate its internal memory if its internal capacity is too small for adding a new element, leaving `n` pointing to freed memory.

Leveraging insights from the previous small-scale study, the team rewrote the ownership chapter of the official *The Rust Programming Language* book [23], incorporating questions from the Ownership Inventory and visualizations generated by *Aquascope* [32].

Compared to the unmodified version of the chapter, a statistically significant average improvement of +9% was observed in performance on the Ownership Inventory questions

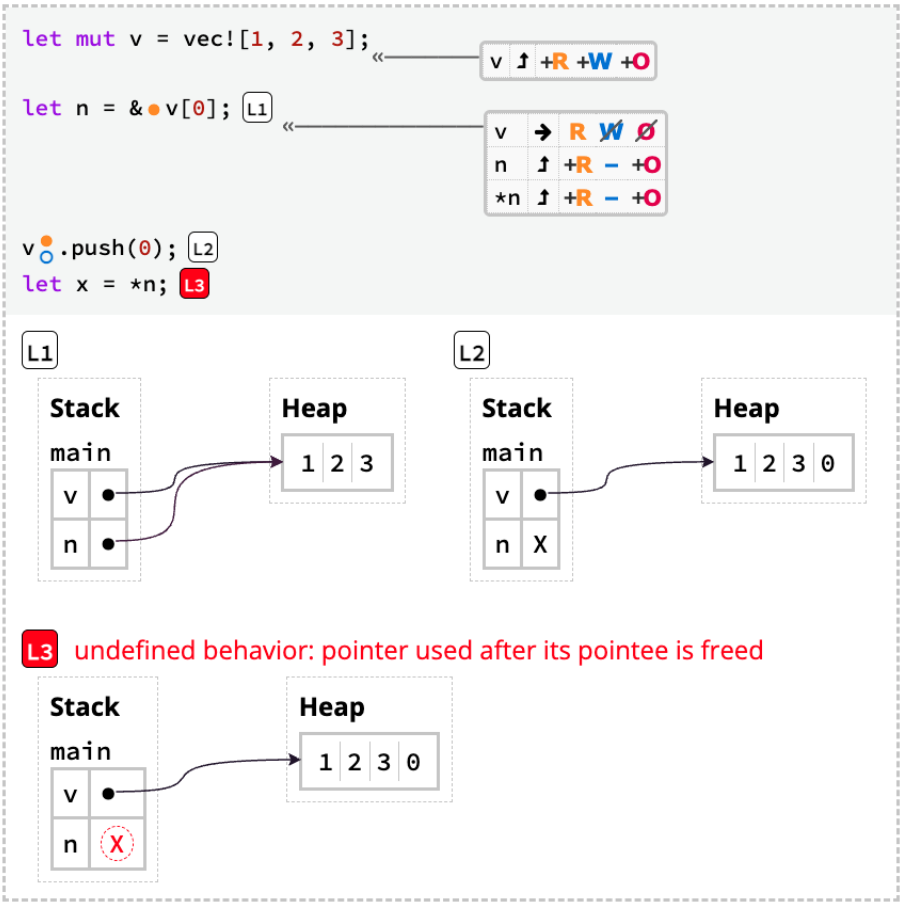


Figure 3.1: Visualization of the internal memory layout behind an unsound Rust program generated by *Aquascope* [30].

among the 177 participants before the rewrite and 165 participants after the modification [30].

Aquascope visualizations are automatically generated by extracting information via a Rust compiler plugin. This allows developers to generate visualizations for their own code snippets [33]. However, the usability of the generated visualizations for larger function bodies is questionable due to potential size and navigability issues.

3.2 Flowistry

Flowistry, another project by the Cognitive Engineering Lab [31], is designed to assist developers during the development process, in contrast to *Aquascope*, which serves more as an educational tool.

Flowistry leverages Rust's ownership and borrowing model to trace the flow of information through code. This enables the identification of function parts that may influence others, by finding a direct flow of data between them. Using this information, *Flowistry* can hide irrelevant code sections that do not affect a selected expression. The underlying hypothesis is that hiding irrelevant parts of a function can help developers focus on the relevant parts, thereby reducing cognitive overhead, especially when dealing with lengthy functions [34].

Figure 3.2 illustrates a function analyzed by the *Flowistry* plugin. As the value `orig_len` is only linked to `set`, the parts of the function related to `other` are grayed out, as they cannot affect `orig_len` in this example.

```rust
fn union(set: &mut HashSet<i32>, other: &HashSet<i32>) -> bool {
  let orig_len = set.len();
  for el in other.iter() {
    set.insert(*el);
  }
  return orig_len != set.len();
}
```

Figure 3.2: Example of code sections relevant to the selection being highlighted by the *Flowistry* editor plugin [34].

Another potential use case for such information flow analysis, enabled by Rust's ownership rules, is the detection of potential leaks of sensitive information. For instance, a direct information flow from a variable containing sensitive data, such as a password, to any output stream could be identified [34].

3.3 RustViz

Figure 3.3: Example of an interactive visualization generated by *RustViz* [35].

RustViz, a project developed at the University of Michigan, is designed to teach Rust's lifetime and borrowing mechanisms to students in their upper-level undergraduate Programming Languages course [35]. Figure 3.3 provides an example of how *RustViz* visualizes these mechanisms. The project represents variables in columns adjacent to the source code and annotates dependencies with arrows between these columns. Hovering over different elements of this graph reveals additional information to the user through hover messages.

In a qualitative analysis involving 303 participants from the course, 47% rated the visualizations as very helpful for understanding Rust's ownership and borrowing, while 46% found them helpful. Despite the generally very positive reception, some participants reported difficulties in following the visualizations at times due to unclear line types. Others found the separation between the source code and the visualization graph bothersome, as it necessitated frequent shifts in focus from left to right [35].

One challenge with *RustViz*'s visualization approach is its scalability to larger programs. The introduction of more variables adds new columns to the graph, complicating the interpretation of the resulting visualizations. Additionally, visualizing ownership in source code that includes branching control flow is problematic, as it disrupts the linear ownership flow in a column.

Currently, *RustViz* is primarily an educational tool, as generating such visualizations requires manual annotation of the example source code. Consequently, the creators of these

visualizations already need a solid understanding of Rust's ownership mechanisms, and students cannot automatically generate visualizations for their own code examples.

3.4 Graphical depiction of ownership and borrowing in Rust

Various visualization ideas have emerged within the Rust community to simplify the understanding of Rust's ownership and borrowing model. However, most of these proposals have either not been implemented yet or are no longer maintained. Despite the lack of formal validation of their effectiveness, these approaches served as valuable references during the development of the visualizations proposed in this thesis.

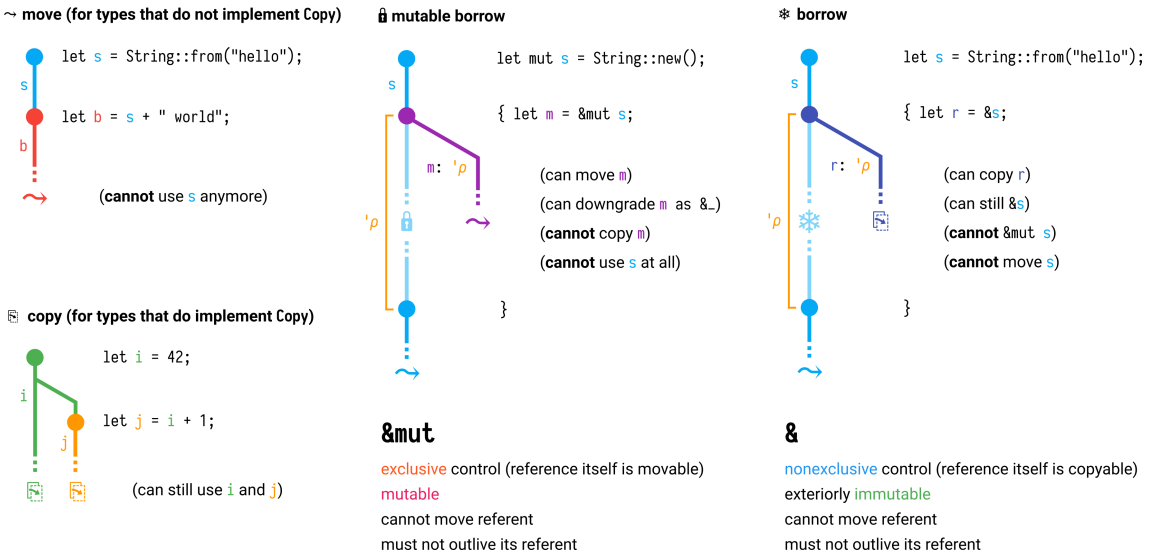


Figure 3.4: Graphical view on Rust's ownership and borrowing mechanisms proposed in a blog post by P. Ruffwind [36].

Figure 3.4 illustrates a graph-based view of Rust's ownership model, as proposed in a 2017 blog post [36], which also served as a visual inspiration for *RustViz*. This blog post only introduced these visualizations as a way to think about the concept of ownership and lifetimes spatially, but never intended to generate such visualizations for a given Rust example.

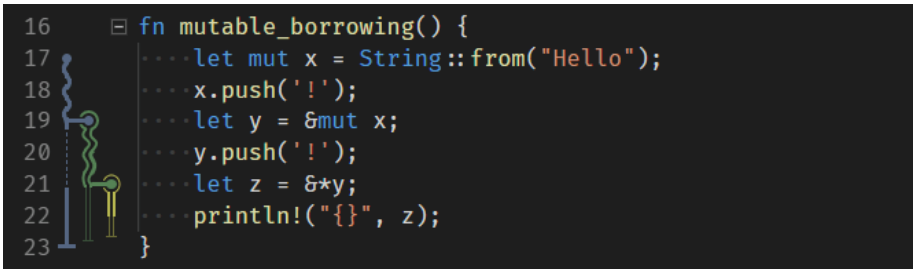


Figure 3.5: Potential adaption of the visualization ideas proposed by [36] to the Integrated Development Environment (IDE) [37].

Figure 3.5 presents another visualization mock-up inspired by Ruffwind's blog post [36]. This mock-up focuses on the potential integration of such visualizations into the Integrated Development Environment (IDE) for practical applications. The author of the blog post [37] also discusses potential challenges in scaling the annotations to more complex functions and properly handling more complicated constructs like closures or NLL.

3.5 RustLifeAssistant

In the bachelor theses [38] and [39], the authors address a distinct challenge associated with writing Rust programs: understanding complex reference dependencies and corresponding borrow checker errors. They argue that the compiler's error messages, particularly those related to borrowing, can be difficult to comprehend and may not provide all the necessary information to understand the core issues leading to borrow checker errors.

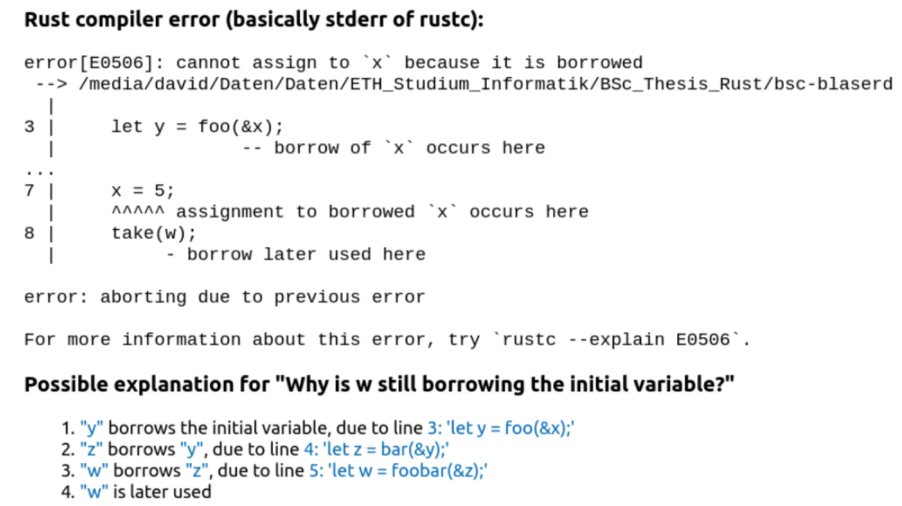


Figure 3.6: Rust borrow checker error annotated with an explanation from *RustLifeAssistant* [39].

To address this, they propose the extraction of a complete lifetime dependency graph, which includes the unsound dependency, for debugging error messages [38]. This graph is then used to generate a causal chain in textual form leading to the error, which is displayed to the developer using an IDE extension named *RustLifeAssistant* [39]. Figure 3.6 presents a borrow checker error output from Rust compiler (`rustc`), annotated with additional error information generated by the plugin.

The authors have successfully automated the process of extracting dependency graphs from the borrow checker's output and integrating the resulting output into the IDE. However, there were instances where the analysis failed to yield usable output. Moreover, the tool's usability has not yet been evaluated on a larger code base in real-world scenarios, but has only been tested on a set of example problems.

3.6 Rust Error Visualizer (REVIS)

The Rust Error Visualizer (REVIS) is another IDE extension designed to enhance the understanding and debugging of lifetime-related borrow checker errors [40]. Unlike *RustLifeAssistant*, discussed in the last Section 3.5, REVIS does not augment the compiler's error messages with additional information. Instead, it focuses on improving the comprehensibility of existing errors for developers by transforming textual error messages into visual annotations directly within the source code.

Figure 3.7 illustrates how REVIS directly annotates borrow checker errors within the developer's IDE, as opposed to the compiler's textual output of the same error shown in Figure 3.8. By integrating error annotations directly into the source code, developers can easily map the compiler's error log to specific locations in their code, eliminating the need to switch focus between the console output and the source code. However, it is important

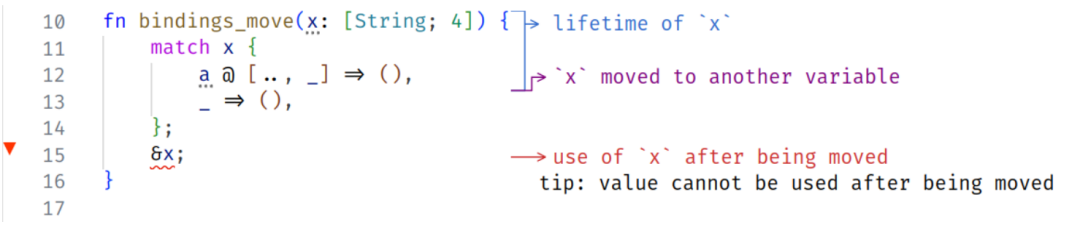


Figure 3.7: Lifetime error visualized by REVIS [40].

```
error[E0382]: borrow of moved value: `x`
  --> src/rust/bindings_after_at_or_patterns_slice_patterns_box_patterns.rs:15:5
   |
10 | fn bindings_move(x: [String; 4]) {
   |                  - move occurs because `x` has type `[String; 4]`, which does not implement the `Copy` trait
11 |     match x {
12 |         a @ [.., _] => (),
   |         - value moved here
...
15 |     &x;
   |     ^^ value borrowed here after move
   |
help: borrow this binding in the pattern to avoid moving the value
   |
12 |         ref a @ [.., _] => (),
   |         +++
```

Figure 3.8: Text-based error log of the error visualized in Figure 3.7.

to note that REVIS currently only supports annotating a limited subset of borrow checker errors.

In their evaluation, the authors measured the time taken by participants to resolve various Rust errors. However, due to limited participation of six students (of which only 3 used REVIS) in their evaluation, no definitive conclusions regarding the impact of REVIS on error-fixing time could be drawn. Unexpectedly, time spend on lifetime-related errors was low, at only 3.4% of total error-fixing time, however, this may be attributed to the limited participation, or low complexity of the errors.

4 Approach

A significant challenge encountered in previous works trying to visualize lifetimes, is that the notion of variable's lifetime does not map naturally to the conventional text-based representation of source code. This issue becomes particularly pronounced in the context of branching code, where finding an intuitive visualization approach proves difficult.

Consider the example provided in Listing 4.1. When the code is read linearly from top to bottom, the variable `x` becomes alive in line 1 and remains so until it is conditionally moved in the `if`-branch in line 4, rendering `x` inaccessible thereafter. However, in the `else`-branch in line 7 accessing `x` remains valid, despite its occurrence 'after' the move in line 4. While it may be relatively straightforward to understand why `x` regains accessibility in this instance, the same might not hold true for more complex, deeply nested examples.

```
1 let x = String::from("Hello");
2
3 if .. {
4     take(x);            // x is moved here
5     // can not access x anymore for the rest of the block
6 } else {
7     print!("{}", x);    // x is back 'alive' here
8 }
```

Listing 4.1: Example of conditionally moving a variable.

In general, the liveliness of a variable is associated with a span in the program's execution graph, rather than a sequence of consecutive source code lines. This discrepancy poses challenges not only with branching code, but also when the evaluation order deviates from the top-to-bottom reading order that most people intuitively follow. Listing 4.2 provides an example of such a potential mismatch in evaluation ordering.

```
1 let mut y = String::from("Hello");
2 y = {                                // y is reassigned
3     println!("{y}");                 // original value of y is accessed
4     String::from("World")
5 };
```

Listing 4.2: Mismatch between line ordering and evaluation order.

When generating visualizations for user-provided code snippets, another consideration is that no assumptions can be made about the formatting of the source code. Given that the Rust syntax permits arbitrary whitespace between tokens [41], a function's entire source could potentially be written in a single line. Such source code would need to be formatted into a more human-readable layout before intuitive visualizations of ownership and lifetimes can be generated. Rust already provides an official tool for formatting code according to a style guide, known as `rustfmt` [42]. While this addresses some of the aforementioned issues, it does not provide a solution for complex branching code.

In light of these challenges, this thesis proposes a custom approach for rendering and laying out Rust code, which is more conducive to displaying spans in a given program's execution graph. This renderer is not based directly on the code's textual representation, but rather on a higher-level abstraction of the code generated during the compilation process. A more detailed explanation of the compiler's internal workings is provided in the following Section 4.1, and details about the actual rendering and layouting process can be found in Section 5.4.

4.1 Rust Compiler Overview

The Rust compiler, also known as `rustc`, transforms text-based source code into various Intermediate Representations (IRs) during the compilation process. These transformations are necessary, as working directly with text-based source code for compilation is very inconvenient. Each IR is designed for a specific purpose in the analysis and compilation of the source code [43]. The process of transforming one IR into the next is referred to as 'lowering', as each successive IR is 'lower-level', closer to the final machine code.

Figure 4.1 provides a high-level overview of the pipeline that Rust source code goes through inside the compiler until a final runnable program is emitted.

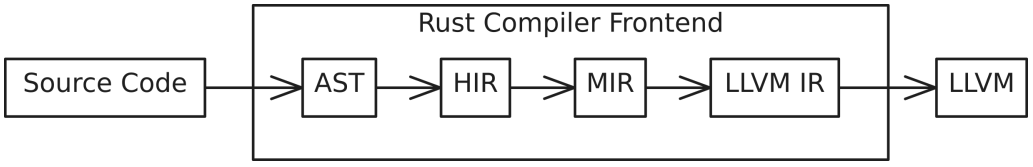


Figure 4.1: Schematic overview of the internal pipeline in `rustc`.

It's important to note that `rustc` is only a so-called compiler frontend, which means it uses a compiler backend to emit a platform-specific runnable binary. The Rust compiler can be used with multiple different compiler backend projects, with LLVM being the most common one [44]. For the purposes of this thesis, only the compiler frontend is relevant, so further details about compiler backends are omitted here.

The IRs mentioned in Figure 4.1 can be summarized as follows [43]:

- Abstract Syntax Tree (AST): The generation of AST involves lexing the source code into a stream of tokens, which are then parsed according to the Rust syntax definition. In general, the AST is still rather close to the original source code. However, in the case of Rust, lowering to AST also deals with macro expansion and conditional compilation.
- High-Level Intermediate Representation (HIR): Structurally, HIR is still very similar to Rust source code. However, some language features are desugared to a slightly lower-level form. This reduced feature set makes HIR more convenient for further analysis compared to AST, which is why HIR is the main data structure used by the Rust compiler. Additionally, HIR is used for Rust's type inference and type

checking. HIR with fully inferred type information is also referred to as typed HIR (or THIR) in the compiler.

- Mid-level Intermediate Representation (MIR): At the level of MIR, it becomes very hard to recognize the original Rust code. It is a heavily simplified form of HIR, which only relies on a very basic set of minimal instructions. This is the IR used for performing data flow analysis on the code and, most notably, the borrow checker [45].
- Code Generation IR: This is the IR used by the specified compiler backend (e.g., LLVM). After some optimizations have been run on MIR, it is lowered once more to the backend IR and passed on for final code generation.

For the visualization, both AST and HIR were potentially viable options. However, due to the reduced feature set, HIR was found to be easier to work with and thus was chosen as the basis for the rendering. Basing the visualizations on a lowered IR of the source code introduces the problem that some language constructs are desugared into a simplified form, e.g., macro expansion, `for`-loops, or the `?`-operator. However, both AST and HIR have some desugaring applied to them, so steps had to be taken to re-sugar these constructs, regardless of which of the two IRs had been chosen. The process of re-sugaring the output is described further in Section 5.4.6.

4.2 Code Analysis

Before generating visualizations of ownership and borrowing, the necessary information must first be extracted from a given Rust code snippet. One approach could be to directly use the output from `rustc` using a compiler plugin. This ensures that the information used for generating the visualizations is the same as that used during compilation. However, the Application Programming Interface (API) for interacting with `rustc` (`rustc_interface`) is still unstable, necessitating the use of an unstable, nightly build of the compiler, making the setup process rather inconvenient for users [46]. Most of the projects mentioned in Chapter 3, supporting automatic code analysis, such as *Flowistry*, *Aquascope*, and *RustLifeAssistant*, have chosen this approach.

Instead of basing the code analysis on the output from `rustc`, this thesis extracts information about the source code using `rust-analyzer` (RA), the official language server of the Rust programming language [47]. A language server is a compiler frontend that provides abstract semantic information about the source code to the IDE via the open Language Server Protocol (LSP) standard. This allows developers to access useful utilities, such as 'Go to Definition' or 'Find All References', via their IDE, without the IDE having to reimplement these utilities for each language [48].

RA is a partial reimplementation of `rustc`, excluding any of the code generation aspects. This reimplementation of the compiler has been heavily optimized for incremental code updates and on-demand queries. An older iteration of a language server for Rust, known as Rust Language Server (RLS), utilized the `rustc` output more directly. However, this resulted in slower response times, leading to a suboptimal developer experience [49]. Implementing and maintaining two separate compiler frontends, `rustc` and RA, leads to inefficient utilization of development resources. Therefore, the plan is to move as much code as possible to libraries, which can then be shared between the two compiler frontends. However, this process is still ongoing.

The tool developed in this thesis is aimed at assisting developers during development, so the quicker response times of RA should be advantageous. Additionally, a future goal is to integrate the proposed visualizations directly into the IDE. This makes basing the analysis

on RA the obvious choice, as it would already be running in the background and would only need to be extended with the proposed analysis.

The main drawback of basing the analysis on RA is that it does not yet offer full feature parity with `rustc`. This is also the primary reason why existing visualization tools rely on `rustc` instead of RA. Most notably, it does not implement any of the borrow checker analysis out of the box. To provide error diagnostics about borrow checker errors, RA currently defers to `rustc` in a background thread via a so called *flycheck*, highlighting any text spans marked by the compiler in the IDE [50].

However, this compiler output does not contain all the information required for the visualizations. This means that all the ownership and reference tracking must be manually extracted from the MIR representation of the source code. Recreating a complete implementation of the Rust borrow checker would have been far beyond the scope of this project. However, only a small subset of the analysis is required to generate useful graphics for tracking ownership and lifetime information through the program. This means that the visualizations will show *how* the Rust compiler thinks about the source code, but will not detect or mark any borrowing-related errors. For this purpose, the `flycheck` mechanism of RA could be reused, but this was not implemented as part of this thesis. A full overview of the limits of this handwritten analysis is summarized in Section 6.1.1.

4.3 Project Outline

For the purposes of this thesis, all the analysis and visualizations are implemented as a standalone application, to allow for faster development iterations during the initial evaluation phase. However, wherever possible the code is structured in a way, that would facilitate integrating the analysis and visualizations directly into the IDE in the future, provided they prove useful.

The application can load any Rust project and extract a tree of all contained modules and functions using RA. Users can select a function from this tree for analysis. Upon successful analysis, an interactive visualization of the selected function is displayed, encoding information about the lifetimes of references and the scopes of local variables.

Further details on the implementation of this analysis are provided in Section 5.2, while Section 5.4 discusses the custom rendering technique used for visualizing the results.

In addition to the standalone desktop application, a simple browser-based viewer application has been developed. This viewer solely renders previously analyzed function bodies in a browser window, eliminating the need for any installation on the user's computer. This approach simplifies the testing of visualizations and facilitates feedback collection during the evaluation stage of this thesis.

5 Implementation

At a high level, the application initiates by requesting the HIR body of the targeted function from RA (refer to Section 4.1). Following this, it conducts a basic data flow analysis on all MIR bodies within this function. This analysis results in a graph depicting assignments, usages of variables, and dependencies between them.

This graph represents relationships within the MIR body. However, for visualization, these relationships need to be relative to the HIR function body, therefore, they are translated back from MIR to HIR. Once translated back to HIR, the information from the data flow analysis is utilized to identify spans where a given variable or reference is alive.

The renderer then attempts to render the function based on its HIR, annotated with these liveliness spans, in a manner that is both readable and useful for the developer.

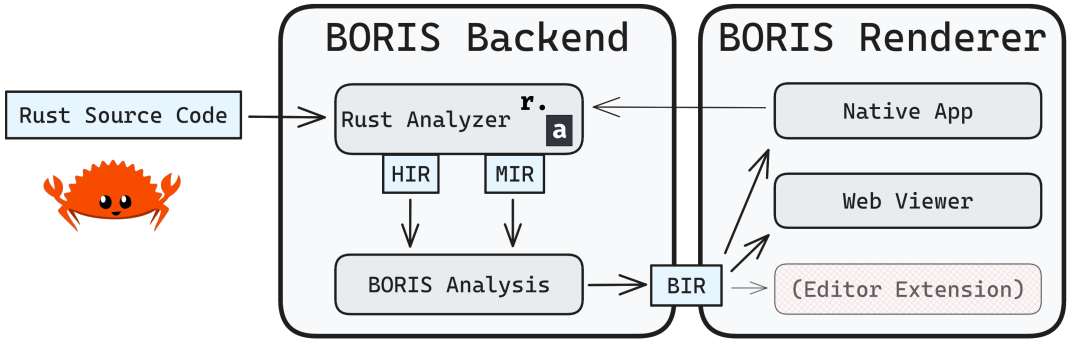


Figure 5.1: High-level structure of the BORrow vISualizer (BORIS) application.

The application developed in this thesis is referred to as BORIS for simplicity. Figure 5.1 illustrates the high-level structure of this application.

5.1 BORIS Intermediate Representation (BIR)

Listing 5.1 presents a simplified representation of how RA internally represents the HIR of a function body. Essentially, it is a tree of `Expr`-nodes, with `body_expr` serving as the root. Each `Expr`-node signifies a core operation of the Rust language. The current implementation encompasses 34 distinct expression types, of which only a select few are listed here.

In the context of a HIR-`Body`, all expressions are stored within an `Arena<Expr>`. This `Arena<Expr>` essentially serves as a wrapper around a basic `Vec<Expr>`. The key distinction lies in the indexing operation: an `Arena` can only be indexed using an `Idx<Expr>` (alias `ExprId`) which is basically a strong-type wrapper around a basic `u32` integer value. This restriction enhances type safety during the indexing process.

```
pub struct Body {
    pub exprs: Arena<Expr>,
    pub pats: Arena<Pat>,

    pub params: Vec<PatId>,
    pub body_expr: ExprId,
    ...
}

pub enum Expr {
    Path(Path),
    Let { pat: PatId, expr: ExprId },
    Match { expr: ExprId, arms: Box<[MatchArm]> },
    UnaryOp { expr: ExprId, op: UnaryOp, },
    ...
}

pub enum Pat { ... }
```

Listing 5.1: Simplified definition of a HIR body from the RA source.

In addition to expressions, the HIR body also includes a separate field for patterns, which are employed during pattern matching (e.g., `if let <pattern> = <expr> {}`), defined in `Pat`. However, this separation between patterns and expressions poses challenges when defining liveliness spans within the body (refer to Section 5.3.5). To address this and some other issues mentioned later, BORIS introduces a separate IR, derived directly from HIR, termed BORIS Intermediate Representation (BIR).

BIR combines the expressions and patterns from HIR into a unified structure, simplifying the traversal process. It also implements some other minor modifications to HIR to facilitate the visualization process. For instance, during the construction of BIR, all paths and names from HIR are resolved into a printable format for rendering. Consequently, the BIR of a function is fully self-contained and does not depend on the current state of RA, enabling its storage and rendering at a later stage without any dependency on RA. This approach, however, renders BIR less memory-efficient compared to HIR, but this trade-off is considered acceptable given the enhanced flexibility and simplified rendering it offers.

In the source code, the structure that amalgamates expressions and patterns from HIR is denoted as a BIR-`Def`, an abbreviation for BIR-body *definition*. These `Def`s are also stored within an `Arena<Def>`, and indexed by a `DefId`. Using this `DefId` parent-`Def`s can refer to their child-`Def`s, spanning a `Def`-tree for each function body.

5.2 MIR Analysis

The primary data flow analysis, akin to the `rustc` borrow checker, is conducted at the MIR-level, due to its more simplified structure in comparison to HIR (refer to the subsequent Section 5.2.1 for more details). For each HIR body of a single function, there may exist one or more MIR bodies:

- One corresponding to the main function body.
- One for each contained closure, which will be further elaborated in Section 5.3.2.

Despite minor differences, the core methodology for analyzing the MIR bodies remains largely consistent between bodies originating from the main body or a closure.

5.2.1 MIR Body Structure

The main characteristic, making the Mid-level Intermediate Representation (MIR) useful for data flow analysis, is that it does not allow nested expressions [45]. This restriction reduces the number of cases that need to be addressed during the analysis. For instance, a nested expression such as:

```
x = (a + b) + c
```

would instead be represented as two separate statements:

```
tmp = a + b
x = tmp + c
```

Furthermore, the MIR is structured as a Control Flow Graph (CFG) of the program. This implies that a branch in program execution, for example, caused by an `if`- or `match`-expression, is also modeled as a branch in the MIR-graph. Specifically, MIR is constructed from a set of interconnected `BasicBlock`s, as depicted in Listing 5.2. Each `BasicBlock` comprises a sequence of `Statement`s, which are guaranteed to be executed sequentially, and one `Terminator` at the end. The `Terminator` may link to multiple other `BasicBlock`s to be executed subsequently based on some condition.

```
pub struct MirBody {
    pub basic_blocks: Arena<BasicBlock>,
    pub locals: Arena<Local>,
    pub start_block: BasicBlockId,
    ...
}

pub struct BasicBlock {
    pub statements: Vec<Statement>,
    pub terminator: Terminator,
    ...
}

pub struct Local { pub ty: Ty }
```

Listing 5.2: Simplified definition of a MIR body from the RA source.

Whenever possible, each `Statement` and `Terminator` links back to a location in HIR from where it originated. This linkage will be crucial in Section 5.3.4, when mapping found dependencies in MIR back to HIR.

5.2.1.1 Locals

Statements and terminators operate on a set of `Local`s, each of which represents a specific region in memory of a given Rust type. Typically, after lowering to machine code, a local is backed by a location in stack memory. However, some locals may exist only as temporary values in CPU registers. Conventionally, the local with the index 0 corresponds to the return value, while the indices $[1; n]$ correspond to the n parameters of the current function. Some other local indices correspond to variable bindings named in the source code, while others are temporary values required due to the flattening of nested expressions.

5.2.1.2 Statements

Among the various types of statements, the most significant for the subsequent data flow analysis is the `StatementKind::Assign(Place, Rvalue)`, which assigns a `Rvalue` to a specified `Place`.

`Place`

A `Place` identifies a specific location in memory and is represented as a local with several projections applied to it. A projection may include (but is not limited to):

- `Field(Either<FieldId, TupleFieldId>)`: A sub-field of the current type.
- `Deref`: Dereferencing a reference or pointer to the memory location it points to.
- `Index(LocalId)`: A specific index of an array, where the index is provided by another local.

For instance, the following expression from Rust code:

```
*myArray[42].x
```

would be represented as a `Place` in MIR by the local corresponding to `myArray`, with the following projections applied sequentially:

1. Index `myArray` by a local containing the value 42.
2. Dereference the reference stored in the array.
3. Project to the field `x` of the dereferenced value.

`Rvalue`

The `Rvalue` encapsulates the majority of operations available in MIR. Some of the most common `Rvalue` types are shown in Listing 5.3.

```
pub enum Rvalue {
    Use(Operand),
    Ref(BorrowKind, Place),
    BinaryOp(BinOp, Operand, Operand),
    UnaryOp(UnOp, Operand),
    Discriminant(Place),
    Aggregate(AggregateKind, Box<[Operand]>),
    ...
}
```

Listing 5.3: Excerpt of the `Rvalue` definition from the RA source.

An `Operand` can either refer to another `Place` or a constant value. The latter will be less relevant during the analysis, as they generally remain unchanged. From this definition of `Rvalue`, it is evident that each statement may read from, recombine, or reference other places. For instance, `Rvalue::Aggregate(kind, operands)` combines the provided operands into a more complex type such as an array, tuple, closure, or an algebraic data type, such as a `struct` or an `enum`. More information about the functionality of each specific `Rvalue` type is provided in the `rustc` documentation of the MIR [51].

It is important to note that `Rvalue` explicitly does not support calling a function, as the called function could panic, which may necessitate special unwinding logic at the call site.

5.2.1.3 Terminators

The statements inside a `BasicBlock` are executed unconditionally in sequence. To allow for conditional execution, multiple `BasicBlock`s must be linked together. For simplicity, MIR only allows branching to another `BasicBlock` as the last operation, the `Terminator` of the current block.

The most important `TerminatorKind`s, used during normal function execution, are shown in Listing 5.4. Most of the remaining `TerminatorKind`s are related to unwinding after a panic and executing custom drop logic for a given `Place`, both of which are not essential for the following analysis. Detailed information can also be found in the official `rustc` documentation [51].

```
pub enum TerminatorKind {
    Goto { target: BasicBlockId },
    SwitchInt { discr: Operand,  targets: SwitchTargets, },
    Call {
        func: Operand,
        args: Box<[Operand]>,
        destination: Place,
        target: Option<BasicBlockId>,
        cleanup: Option<BasicBlockId>,
    },
    Return,
    ...
}
```

Listing 5.4: Excerpt of the `TerminatorKind` definition from the RA source.

When a `TerminatorKind::Goto { target }` is encountered, execution simply continues in the `target-BasicBlock`.

All conditional execution is represented as a `TerminatorKind::SwitchInt { discr, targets }` in MIR, where `discr` is required to contain an integral value. `targets` uniquely maps the discriminant value to a `BasicBlockId`, where execution is continued when the value matches. When no value matches, execution continues at a specified `otherwise-BasicBlockId`.

As previously mentioned in Section 5.2.1.2, all function calls are represented as a `TerminatorKind::Call { func, args, destination, target, cleanup }`, as execution may have to be deferred in case the called function panics. However, if the called function returns normally, execution is continued at the `target-BasicBlockId`, with the returned value assigned to the `destination-Place`.

The terminators link the basic blocks together to form a connected, directed graph with a fixed entry point, often referred to as the CFG of a given function. Note that this CFG may contain cycles, in cases where the HIR contained `loop{}`-expressions. Figure 5.2 shows a schematic representation of such a CFG which would be generated from the following source code:

```
let vec = ...;
for elem in vec {
    process(elem);
}
```

As MIR is a significantly simplified version of the original code, it tends to become incomprehensible to humans, even for small functions.

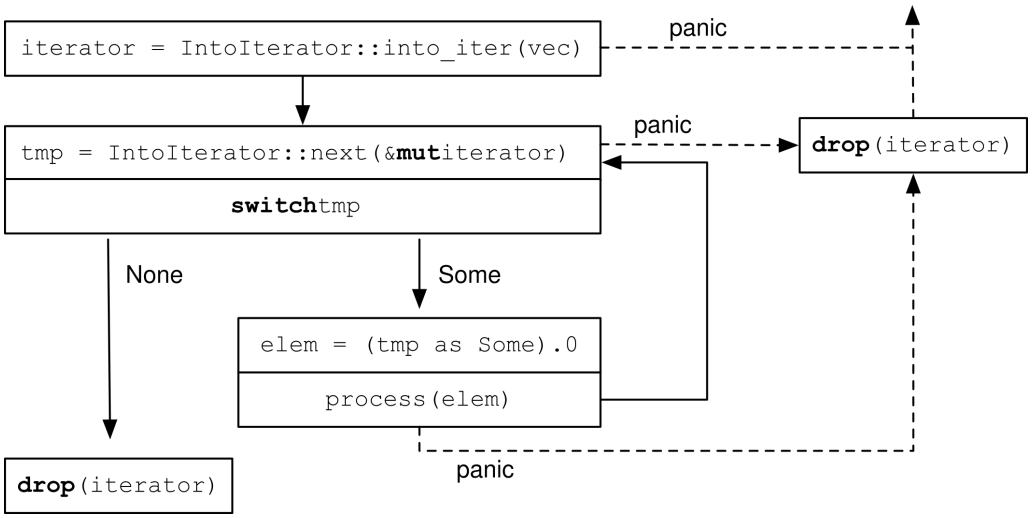


Figure 5.2: Schematic example of a MIR-CFG [45].

5.2.2 Move Paths

The initial step of the analysis involves tracking the initialization and ownership transfer of locals in the MIR body. Given that Rust permits partial initialization and moves of locals, tracking the initialization state at the local level is insufficient. Instead, it must be tracked at the `Place` level (refer to Section 5.2.1.2). This part of the analysis is heavily influenced by `rustc`'s implementation, which employs so-called `MovePaths` for this purpose [52].

```
pub struct MovePath {
    pub next_sibling: Option<MovePathIndex>,
    pub first_child: Option<MovePathIndex>,
    pub parent: Option<MovePathIndex>,
    pub place: Place,
}
```

Listing 5.5: Simplified version of the `rustc` `MovePath` definition.

Listing 5.5 provides the `rustc` definition of a `MovePath`. Each `MovePath` is uniquely mapped to a `Place` in the MIR body. However, not every possible `Place` is also represented by a `MovePath`.

Prior to the main analysis, `rustc` traverses the entire MIR body and identifies all `Places` that are read from or written to. For each of these `Places`, where a move from or to would be legal, a `MovePath` is generated, identifiable by its unique `MovePathIndex`. For instance, it would be illegal to move a `Place`:

- that lies behind a reference[1], as this would leave the reference pointing to invalid memory, which is prohibited by Rust's borrowing rules. Thus, no `MovePath` is generated for `Places` containing a `Deref`-projection.
- that lies behind a specific index of an array[2], thus no `MovePaths` are generated for `Places` with `Index`-projections.

[1] It is possible to 'move' data out of a unique reference via functions like `std::mem::replace`[53], as these functions are designed to not interfere with Rust's ownership model.

[2] As an array may contain thousands of elements, and the index value of the access may not be known at compile time, Rust disallows moving single elements from an array so far, even though this may be safe to do in some cases.

Avoiding the generation of MovePaths for Places that cannot be moved reduces the overall overhead of the move analysis.

```
struct MyStruct<'a> {
    a: String,
    b: &'a String,
}
```

For instance, given a MIR local x of type MyStruct, a MovePath would be generated for x itself, x.a, and x.b, but not for *x.b, as moving the value referenced by x.b is disallowed.

5.2.3 Place Inhabitedness Analysis

This section outlines the steps required for a simplified analysis, which only checks if all accessed MIR-Places have been initialized before being accessed. This analysis serves as a precursor to the full analysis, which tracks moves and references, discussed in the next Section 5.3. The underlying steps necessary for this reduced analysis are identical to those for the full analysis, but starting with a simplified problem set allows for a more straight forward description of the algorithm.

If the CFG spanned by the BasicBlocks were acyclic, this problem could be solved by sorting the BasicBlocks in topological order [54], and then sequentially traversing them, keeping track of which MovePaths were initialized at the end of the previous block. However, this approach is not generally applicable, as MIR may contain cycles when loop-expressions are used.

```
let text = String::from("Hello");
loop {
    drop(text);
}
```

Listing 5.6: Error: use of moved value 'text'.

In the example shown in Listing 5.6, the value text is successfully moved into the drop function in the first iteration of the loop. However, in the second iteration, the local associated with text would be considered uninitialized, resulting in an error. Therefore, a more sophisticated algorithm is required for the general case, as described below:

1. The analysis begins by collecting all MovePaths of the given MIR body. The current state of each MovePath can be represented as a basic mapping from each MovePathIndex to a bool-flag indicating whether it has been initialized. This mapping is henceforth referred to as MovePathStates.
2. A MovePathStates map is assigned to the beginning and end of each BasicBlock, with every MovePathIndex defaulted to be uninitialized. The only exception being all MovePaths associated with function parameters, which are considered initialized at the entry to the starting block.
3. A queue is created containing all BasicBlockIds, with the starting block as the first entry.
4. The first BasicBlockId is taken from the queue, and all statements and the terminator of the BasicBlock are sequentially traversed, keeping track of which MovePaths are assigned/moved by each statement.
5. After traversing the BasicBlock, the MovePathStates associated with the end of the block are updated.

6. For each `BasicBlock` following the current one, all `MovePathStates` associated with the end of its predecessors are merged. In this context, merging means that a given `MovePath` is only considered initialized if it is initialized in the`MovePathStates` of *all* predecessors. If the merged `MovePathStates` differ from the current `MovePathStates` associated with the block's entry, the entry state is updated, and this block is enqueued for processing, if it is not already in the queue.
7. Repeat everything from step 4 until the queue is empty.
8. Each `BasicBlock` is then traversed once more, starting with the `MovePathStates` associated with its beginning, and updating the state according to each statement encountered. If at any point a `MovePath` is accessed with an unset state flag, the program is unsound, as an uninitialized or moved variable would be accessed.

Steps 4-7 of the above algorithm constitute a fixed-point iteration, as the loop runs until a stable state or fixed-point is reached.

5.3 MIR Dependency Graph

This section elaborates on the construction of a more detailed data flow graph within the function required for the visualization. This graph not only tracks *if* a given `MovePath` is initialized, but also *where* it was assigned. Additionally, the visualization has to be able to track references, or more specifically the lifetime constraints imposed by those references, through the sequence of statements. To achieve this, BORIS transforms the `BasicBlock`-CFG into a more granular graph, which contains nodes for each assignment and usage of a place, with edges between the nodes encoding data dependencies.

From this point, the analysis implemented in this thesis significantly diverges from the `rustc` borrow checker implementation [55] for several reasons:

- The aim of this thesis is not to re-implement a borrow checker capable of *validating* the borrowing rules, but rather to visualize *what* the borrow checker 'sees' when analyzing the code.
- Writing a full borrow checker, with all of its capabilities, is highly complex and considered beyond the scope of this thesis.
- RA does not yet provide all the necessary information for a full custom borrow checker implementation. This point is further elaborated in the following Section 5.3.1.

To construct this MIR dependency graph, BORIS first analyzes each statement and terminator of all MIR `BasicBlocks` individually, translating them into assignment and usage nodes. At the end of this initial step, each `BasicBlock` is transformed into a sequence of `NodeStatements` (see Listing 5.7). Section 5.3.1 provides detailed information on how the different kinds of MIR statements and terminators are transformed into such `NodeStatements`.

```
struct NodeStatement {
    assigned: Vec<(MovePathId, NodeId)>,
    used: Vec<(MovePathId, EdgeKind, NodeId)>,
}
```

Listing 5.7: Definition of a `NodeStatement` used for generating the MIR dependency graph.

An analysis similar to the inhabitedness analysis from Section 5.2.3 is performed using these `NodeStatements`. However, instead of tracking an initialization flag for each `MovePath`, a

set of NodeIds, where a given MovePath was last assigned to, is tracked. Additionally, a set of NodeIds, indicating where the MovePath was last moved, is tracked, which will also allow for basic error reporting as described in Section 5.3.3. Merging of the MovePathStates now involves merging the sets of initialization and moved NodeIds of each MovePath. If the assignment Node-sets of one of the predecessor BasicBlocks is empty, this MovePath is flagged as 'maybe unassigned'.

```
let x: i32;
if ... {
    x = 1;
} else {
    x = 2;
}
print!("{x}");
```

Listing 5.8: Example of x potentially originating from different sources.

In most cases, the set of NodeIds where a given MovePath was last initialized will contain a single element. However, as illustrated in Listing 5.8, in the case of conditional control flow, a MovePath could have been last assigned to by different NodeIds. In Listing 5.8, the value of x printed originates either from the if- or the else-block, depending on the condition evaluated at runtime.

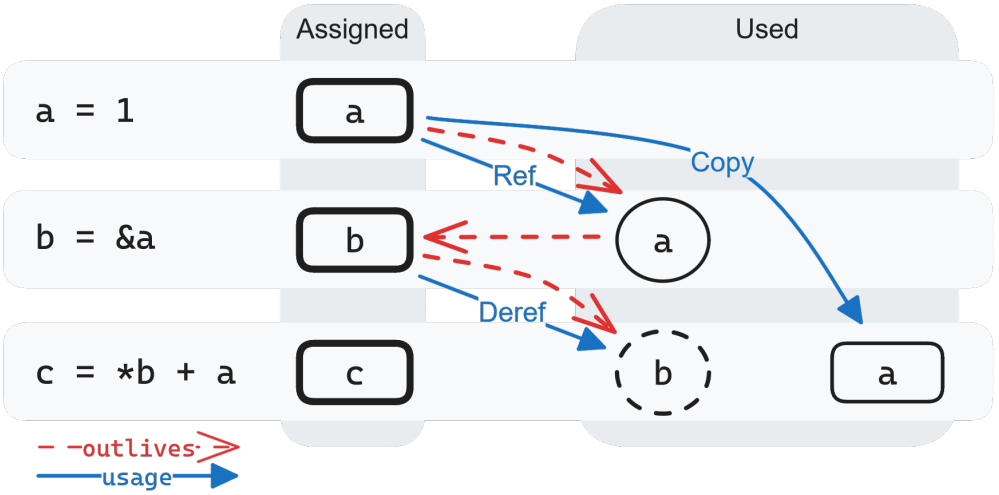


Figure 5.3: Schematic representation of the MIR dependency graph generated by the analysis.

Once a fixed point is reached with the algorithm described in Section 5.2.3, the last assignment or move of each MovePath is known at the beginning of each BasicBlock. The next step involves traversing the NodeStatements of each BasicBlock once more, keeping the MovePathStates updated after processing each NodeStatement. Each time a NodeStatement with usage Nodes is encountered, the MovePathState of the used MovePath allows for the insertion of edges between the initialization Nodes and the Node of the usage.

A brief excerpt of such a connected MIR dependency graph is shown in Figure 5.3 for three basic MIR statements:

```
a = 1;
b = &a;
c = *b + a;
```

The blue arrows represent dependencies between the assignment `Node`s of a `MovePath` and their usages in a subsequent `NodeStatement`. Red arrows represent lifetime constraints between `Node`s containing references, which are explained further in the following Section 5.3.1.

After inserting such edges for each usage into the graph, this MIR dependency graph allows for tracking initializations, copies, moves, and references throughout the entire MIR body. However, before these dependencies can be displayed to the developer, they must be mapped back to the corresponding expressions in the BIR body, which is the topic of Section 5.3.4.

5.3.1 `NodeStatement` Lowering

Before generating the full MIR dependency graph, each individual `StatementKind` and `TerminatorKind` (refer to Section 5.2.1.2) have to be lowered to a `Node`-based representation separately. This process involves lowering all `Place`s occurring in the statement to `Node`s. Regardless of whether a given `Place` is written to or read from, the basic lowering steps are very similar:

1. Identify the `MovePathIndex` of the `Local` associated with the given `Place`.
2. Iterate over the `Place`-projections.
3. For each projection, find the child-`MovePath` and update the current `MovePathIndex`.
4. If a projection (e.g., `Deref`) for which no child-`MovePath` exists is encountered, abort the iteration.

Upon completion of this iteration, a graph `Node` with one of the `NodeKind`s (shown in Listing 5.9) is created, depending on the result and the usage kind of the `Place`.

```
pub enum NodeKind {
    Value,
    Place(Option<BindingId>, bool /* is root MovePath? */),
    Deref {
        mutable: bool,
        binding: Option<BindingId>,
        lvalue: bool,
    },
    Drop,
}
```

Listing 5.9: The different `NodeKind`s for the dependency graph.

If the iteration is aborted due to a `Deref`-projection, a `Deref-Node` is created. The mutability flag of the `Node` depends on the mutability of all the references being dereferenced by the `Place`. If any of the references is not marked as mutable, the entire `Place` is considered immutable. The `lvalue` flag is set if the `Place` occurs on the left side of the assignment, which will become relevant during Section 5.4. The `binding` points back to the named variable in HIR from which the given `Local` originated, if one exists. This is further elaborated in Section 5.3.4.

If the projection iteration completes successfully, it results in the `MovePathIndex` being assigned to or read from. For usages, a basic `Value-Node` is created, and for assigned `Place`s, a `Place-Node` is created.

At the point of lowering the `NodeStatement`s, it is still unknown where a given `MovePath` was last assigned to, so no edges between value source `Node`s and usage `Node`s can be

inserted yet. Instead, each assignment `NodeId` is saved together with the `MovePathIndex` it writes to. Similarly, each usage `NodeId` and `MovePathIndex` is tracked, along with an `EdgeKind` (see Listing 5.10), providing information about how the given usage relates to the value stored at the `MovePath` it points to. The specific `EdgeKind` of each usage-`Node` depends on how exactly the `Place` is used, which is described in more detail at the end of this section.

```
pub enum EdgeKind {
    Ref(bool /* mutable? */),
    Move,
    Copy,
    Deref,
    ...
}
```

Listing 5.10: `EdgeKind`s for expressing the relation between assignments and usages.

It is important to note that each `Deref-Node` is considered a usage of the `MovePath` which is dereferenced, even if the source `Place` is being assigned to. This may seem counterintuitive, but assigning a value to a dereferenced reference does not write to the `MovePath` of the reference directly, but to the memory location the reference points to, as illustrated in Listing 5.11.

```
let mut x = 0;
let ref_x = &mut x;
*ref_x = 42;
```

Listing 5.11: Example of writing to a dereferenced reference.

Once the MIR analysis is complete, it could even be inferred that `*ref_x` writes to the `MovePath` associated with `x`. However, in the general case, such inferences cannot be made. Instead, the `Deref-Node` imposes a constraint on the reference assigned to the `MovePath` of `ref_x`, stating that it has to be valid at the point of the assignment. This is represented as an `EdgeKind::Deref` to the `Deref-Node` in the MIR dependency graph.

These edges inserted between assignment and usage `Node`s allow for tracking the data flow, like moves and copies, between statements. However, the lifetime of a used reference does not stop at the usage location, but may extend to the assignment node of the statement, from where it may be used further.

To track reference lifetimes through the MIR dependency graph, `Node`s representing `Place`s which contain references are annotated with a unique `LifetimeId`. If a `Node` containing a lifetime `'a` is assigned to another `Node` with lifetime `'b`, an `outlives` dependency `'a -> 'b` is inserted into a separate constraint graph. These *outlives* constraints imply that the assigned reference has to 'live' longer, or at least as long, as the reference it is assigned to, to ensure that no reference ever points to 'dead' memory. Combining the MIR constraint graph with this lifetime-outlives-graph allows for tracking lifetime dependencies through the entire MIR body. These outlives-dependencies are represented as red arrows in Figure 5.3, shown in the last section.

When and how these *outlives* constraints are inserted between `Node`s depends on the kind of statement or terminator. For example, for assignment statements containing `Rvalue`s of kind `UnaryOp` or `BinaryOp`, no constraints need to be inserted within the `NodeStatement`, as they always operate on numeric types, which cannot contain references. The operator `Node`s of these `Rvalue`s simply have a `Copy` dependency to the `Node`s where they were assigned.

```
struct S<'a, 'b> {
    x: &'a i32,
    y: &'b i32,
}
```

```
struct S<'a, 'a> {
    x: &'a i32,
    y: &'a i32,
}
```

Listing 5.12: Struct definition as seen by `rustc` (left) compared to BORIS (right).

The `Rvalue` kinds that warrant detailed examination are `Use`, `Ref`, and `Aggregate`. For the `Terminators`, `Call` is of particular interest. Other `Rvalue` and `TerminatorKinds` exhibit behavior similar to the ones described below, hence they are not explicitly discussed here:

- `Use(Operand)`: If the type of the `Operand` implements the `Copy`-trait, the usage is classified as either `EdgeKind::Copy` or `EdgeKind::Move`. When the `Operand` contains a reference, a lifetime constraint is added from the `Operand-Node` to the assignment `Node`.
- `Ref(Place)`: The `Node` representing the `Place` has an `EdgeKind::Ref` dependency on its last assignment `Node`, and it extends the lifetime constraint to the assignment `Node` of the `NodeStatement`.
- `Aggregate(Kind, Operands)`: The current implementation of this `Rvalue` kind behaves similarly to `Rvalue::Use`, but with multiple operators. However, this is not entirely accurate, as Rust allows aggregate types, like `structs` or `enums`, to contain multiple distinct lifetimes through generic lifetime annotations. However, these generic lifetime type annotations are not properly lowered by RA, as they were not necessary for the analysis performed by RA so far [56]. As a result, BORIS currently treats all lifetimes in aggregate types as one, as shown in Listing 5.12. This reduces the overall accuracy of the analysis performed by BORIS, which is further discussed in Section 6.1.1. One aggregate kind, known as a closure, requires some additional special handling and is discussed separately in the following Section 5.3.2.
- `Call { func, args, ..}`: A problem similar to `Rvalue::Aggregate` also occurs for function calls, as lifetime annotations of function calls are not yet properly lowered. However, BORIS performs some basic parsing of the function signature to achieve a more accurate analysis. It identifies occurrences of lifetime annotations in the return type of the function, and only creates *outlives* constraints from arguments that also contain at least one of the return lifetime annotations. In cases where lifetime annotations were elided (see Section 2.3.3.1), *outlives* constraints are inserted for all arguments containing references.

5.3.2 Closures

One important `Rvalue::Aggregate` kind, which has been ignored in the previous section, is closures. Closures are anonymous functions defined within the body of another function and may *capture* variables from the scope of the containing function body. The instructions of such anonymous closure functions are not directly contained within the MIR body of the containing function. Instead, they are within a separate MIR body, akin to a separate function, necessitating separate analysis

```
let mut count = 0;
let mut inc = || {
    count += 1;
    println!("{}", count);
```

```
};
inc();  // 1
inc();  // 2
assert_eq!(count, 2);
```

Listing 5.13: Example of a closure capuring a variable by `&mut`.

For instance, Listing 5.13 illustrates the `inc`-closure capturing the variable `count` via a unique reference. Each subsequent call to the closure increments the counter by 1 and outputs the new value. The rules `rustc` employs to determine what a closure captures and how are not crucial for the analysis, as this is already determined during lowering to MIR, and are not discussed here further [57].

```
struct inc_closure<'a> {
    capture_0: &'a mut i32,
}

impl<'a> std::ops::FnMut<()> for inc_closure<'a> {
    fn call_mut(&mut self, args: ()) -> () {
        *self.capture_0 += 1;
        println!("{}", self.capture_0);
    }
}

impl<'a> std::ops::FnOnce<()> for inc_closure<'a> {
    fn call_once(self, args: ()) -> () {
        self.call_mut(args)
    }
}

// impl<'a> std::ops::Fn<()> for inc_closure<'a> {
//    fn call(&self, args: ()) -> () { /* not possible */ }
// }
```

Listing 5.14: Pseudo-implementation of the `inc`-closure from Listing 5.13 generated during compilation.

Internally, Rust translates each closure into a custom type, with any captured values as fields. For example, Listing 5.14 demonstrates how the `inc`-closure from Listing 5.13 would be roughly represented internally during compilation. This also explains why a closure definition is represented as an `Rvalue::Aggregate` in MIR.

Depending on what a closure does with its captured values, the compiler automatically implements one of three function-traits for it, which will contain the core logic of the closure. If the closure only requires immutable access to its captures, it implements the `Fn`-trait. If it may mutate captured values, it implements the `FnMut`-trait, and if it moves captured values out of the closure, it implements the `FnOnce`-trait. Note that the `Fn`-trait is strictly less restrictive than the `FnMut`-trait, which is less restrictive than the `FnOnce`-trait. This means that a `Fn`-closure can be used in places where a `FnMut` or `FnOnce` closure is required, and similarly, `FnMut` can be used in places where `FnOnce` is required.

In the case of Listing 5.14, the `inc_closure` requires mutable access to the captured values, thus `FnMut` and `FnOnce` can be implemented by the compiler. However, the `Fn` trait cannot be implemented, as it would require implementing the `call(&self, ..)`-method, which would not allow mutating `self`, as it is passed as a shared/immutable reference.

Calling a closure is represented as a basic method call to either `closure.call(args)`, `closure.call_mut(args)`, or `closure.call_once(args)` in MIR, so no special handling is required for the analysis, as this is already handled by the `Call-Terminator`.

To integrate the data flow inside the closure body into the main analysis of the function, the MIR-body of the closure is automatically analyzed upon construction. Essentially, this is just a recursive call to the analysis currently being performed, see Section 5.2. However, the `NodeStatements` of the closure body are embedded into the same MIR dependency graph. When the closure captures a reference from the function's scope, a lifetime constraint to the closure's `self`-parameter `Node` ensures that the lifetime can be tracked into the closure body.

5.3.3 Conflicts

Although error detection and reporting are not the primary objectives of the analysis, certain conflicts can be identified during the generation of the MIR dependency graph with minimal additional effort. For example:

- Use of moved/unassigned value: This conflict arises when a usage `Node` that refers to a `MovePath`, which has not been assigned yet or is marked as moved, is found during the insertion of dependencies between `NodeStatement`s.
- Assign to immutable (reference): This conflict occurs when an assignment is made to a `Node`, where the corresponding `Place` is not marked as mutable.
- Move out of reference: This conflict happens when a `Place` that contains a `Deref` projection is used, and the type of the `Place` does not implement the `Copy`-trait.

In these instances, the affected `NodeId`s are stored adjacent to the dependency graph and can be used later to mark affected expressions during rendering.

5.3.4 MIR to HIR mapping

So far, all data flow and reference dependencies have been expressed in terms of `Node`s referring to locations in the MIR body. However, for rendering, these dependencies need to be relative to HIR, or more specifically, the custom BIR.

As mentioned during the generation of the MIR dependency graph in Section 5.3, MIR statements and terminators link back to the HIR location from which they originated. However, this link is generally not fine-grained enough to identify the exact location of each operand.

```
1 let x = 1;
2 let y = 2;
3 let (z, w) = (x, y);
```

```
1 _x = Const(1);
2 _y = Const(2);
3 _tmp = Tuple(_x, _y);
4 _z = _tmp.0;
5 _w = _tmp.1;
```

Listing 5.15: Example of Rust code (left) being lowered to MIR (right).

For instance, Listing 5.15 shows how a small code snippet would be roughly lowered to MIR. In the current implementation of MIR lowering in RA, the statement in line 3 of the MIR code, `_tmp = Tuple(_x, _y);`, would point to the pattern representing `(z, w)` in HIR. However, it is not straightforward to link to the correct HIR expressions, `x` and `y`, that the MIR operands, `_x` and `_y`, refer to.

This issue could be resolved by saving the original HIR expression or pattern on a per-`Place` level in the MIR, instead of on a `Statement`/`Terminator` level. However, this would necessitate modifications to the MIR-lowering code inside RA. Since maintaining a custom MIR-lowering implementation was not feasible, BORIS employs some best-effort heuristics to map the operands of MIR statements back to their equivalents in HIR. While this approach has proven effective in the current implementation, these heuristics make the mapping susceptible to potential future changes in the MIR-lowering implementation of RA.

These links back to the HIR, or more specifically BIR, are stored on a per-`Node` basis in the MIR dependency graph. However, not all nodes in the MIR graph have a corresponding definition in the BIR. This discrepancy is due to the introduction of local variables during the MIR-lowering process. Upon completion of the MIR traversal, the MIR dependency graph is 'highered' back up to the BIR-level to construct a BIR dependency graph for rendering.

The construction of this BIR dependency graph involves extracting all assignment `Node`s (`NodeKind::Place`) from the MIR graph that correspond to a BIR node. Associated usages can be identified from these assignment `Node`s by traversing the `Move`, `Copy`, `Ref`, or `Deref` edges in the MIR graph. For each of these usages that link back to a definition in the BIR, a corresponding edge is inserted in the BIR dependency graph.

Similarly, references can be tracked using the lifetime constraint graph and transformed into edges in the BIR dependency graph. However, MIR nodes representing locals, introduced during MIR lowering, are not represented in BIR. Therefore, lifetime constraints sometimes need to be tracked across multiple nodes in the MIR graph until a node with a corresponding representation in the BIR is found, allowing for the insertion of an edge in the BIR graph.

5.3.5 BIR Control Flow Sequence

So far, the BIR dependency graph represents the location in the BIR where an initialization occurs and where this value is used. To annotate the span in which a given variable or reference is active, the set of all BIR `Def`-nodes which lie 'between' the initialization and usage are required. In a traditional, text-based representation, this would typically refer to a set of lines in the source code. However, as discussed in Chapter 4, this is insufficient when dealing with branching code or when statements are not separated into new lines.

Instead, BORIS defines these spans as a set of `DefId`s in the BIR. By default, the `rustc` IRs, such as HIR and BIR, are structured as a tree of nodes, where each node may contain child nodes. However, this tree-based representation is not well-suited for gathering all nodes between two other nodes.

To address this, BORIS defines a directed graph of all BIR nodes, where two nodes are connected if they would be evaluated sequentially. This means that nodes in this graph can have multiple predecessors or successors in the case of a branch, as there cannot be an ordering between the different arms of a branch.

Figure 5.4 illustrates this sequential ordering of the tree-based structure, on a tree that would roughly correspond to the expression:

```
1 42 + (7 * 3)
```

With this graph, all nodes occurring 'after' a given one can be gathered by simply walking this sequence-graph and collecting all nodes. Similarly, all nodes before a usage can be collected by walking this sequence-graph in the reverse direction. Checking which nodes

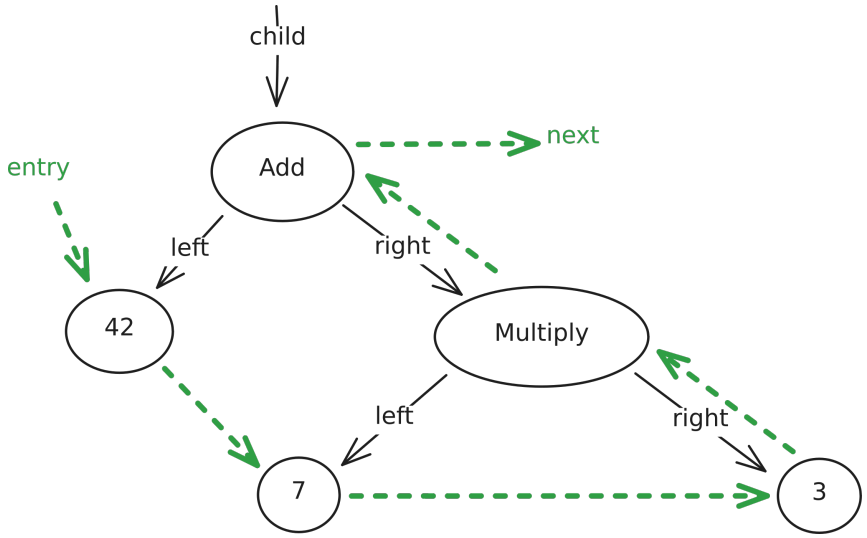


Figure 5.4: Visualization of the sequential ordering of BIR nodes (green dashed arrows), compared to the tree structure (black arrows).

are contained within the set of nodes *after* the assignment `AND` *before* the usage yields all nodes *in between* the two nodes.

Internally, this set of BIR nodes is represented as a `BitVec`[58] containing one bit for each `DefId`. If a `DefId` is included in the set, its associated bit is set to 1; otherwise, it is set to 0. This makes storing and operating on these sets of BIR nodes very efficient. For example, the union of two sets can be calculated via a bitwise `OR`-operation between two sets, and similarly, an intersection can be calculated with a bitwise `AND`-operation.

Therefore, to calculate the liveliness span of a given reference, the BIR `DefId`-set of nodes after the initialization is first calculated. This initialized-set is then intersected with the union of all `DefId`s which lie before the usages of this reference, yielding a set of all `DefId`s the reference is active for.

5.4 Rendering

This section discusses the rendering of the BIR definition, which was generated and annotated in the previous sections. One of the main challenges of the rendering process is finding a way to communicate the borrowing and ownership information, as seen by the borrow checker, to the user. To the borrow checker, the region in which a given variable or reference is alive appears as a continuous partition of the program. However, when mapping to a line-based source code representation, these regions may not be contiguous anymore in cases of branching code. This could limit the usability of the visualizations in cases of more complex code snippets.

Instead, this thesis proposes a novel layout approach for the Rust source code, which is based on the evaluation order of the instructions. It displays parts of complex expressions from top to bottom, depending on their ordering during execution. In this visualization approach, conditional execution flow is drawn next to each other, instead of beneath each other, indicating that the code inside the branches is executed mutually exclusively instead of consecutively.

This approach is very similar to diagram-based rendering of CFGs, akin to the MIR-CFG rendering shown in Figure 5.2. The problem with such diagrams is that they tend to become rather large and hard to follow at a glance for more complicated programs.

Additionally, experienced developers become well-adjusted to reading text-based source code representations in a non-linear, execution-order-oriented manner [59], thus altering the code layout too much would likely have adverse side effects on readability.

The rendering algorithm, described in this section, aims to maintain the core source code layout that developers are already adjusted to, with minimal alterations to allow for a more intuitive annotation of the variable lifetime scopes.

An additional requirement on the renderer is that it has to be able to respond to the user's input, which implies that:

- The rendering needs to be able to adjust to some changing state.
- Some mechanism for mapping user input back to the BIR representation is required.

The implementation of the BORIS renderer is based on the `egui` [60] library, an immediate mode User Interface (UI) library for Rust. However, the rendering of the BIR itself is implemented with a custom layouting algorithm, only utilizing `egui`'s core cross-platform primitive and text rendering capabilities, as complex layouting is one of the inherent weaknesses of immediate mode UI frameworks [61].

5.4.1 Recursive BIR Rendering

The BORIS renderer operates directly on the tree-based BIR representation of the program, rather than the source text. This tree-based structure lends itself for a recursive implementation of the renderer, as rendering a parent node also requires rendering all of its child nodes.

A key challenge in implementing such a recursive renderer is that the parent node does not know how much screen space to allocate for rendering its children, as they in turn may contain multiple child nodes. This issue can be addressed through a two-pass rendering approach: first determining the required size of each node from the leaves up to the root node, and then rendering from the root downward. However, this solution implies that some calculations must be executed and implemented twice: once for determining the required size, and once during the actual rendering.

While the performance overhead of these repeated calculations should be negligible in most cases and thus not a primary concern, separating the sizing calculations from the core rendering logic has proven to be error-prone and time-consuming to implement, especially during the initial development phase. Therefore, a buffer-based layouting approach has been implemented.

In this approach, instead of rendering directly to the screen, each node appends one or multiple `DrawCall`s to a buffer and returns the `DrawCallId` of its top-level `DrawCall` to the parent node. A simplified definition of a `DrawCall` is provided in Listing 5.16. The parent node can then combine the `DrawCallId`s of the child nodes, possibly add its own `DrawCall`s to the buffer, and return the new top-level `DrawCallId` up to its parent node.

This process is repeated recursively until the root node of the BIR body returns a root `DrawCallId`, which encapsulates the rendering of the entire function body. Only after all `DrawCall`s are collected in the buffer is the entire buffer drawn to the screen in one step, by recursively walking the buffer from the root `DrawCallId` down to its children. More details on how the BIR nodes are translated to their corresponding `DrawCall`s is given in Section 5.4.3.

```
1 pub struct DrawCall {
2     pub kind: DrawCallKind,
3     pub size: Vec2,
```

```
}

pub enum DrawCallKind {
    // primitive
    Text(Arc<Galley>, Color32),
    Rect(Color32, Rounding),

    // compound
    Inline(Box<[RelativeDrawCallId]>),
    Branch(Box<[RelativeDrawCallId]>, ...),
    Sequential(Box<[RelativeDrawCallId]>),
    ...
}

pub struct RelativeDrawCallId {
    pub id: DrawCallId,
    pub offset: Vec2,
}
```

Listing 5.16: Simplified `DrawCall` definition of the renderer implementation.

As the root `DrawCallId` encapsulates the drawing of the entire function body, the exact size needed for the rendering can now be read directly from the `size` field of the root `DrawCall`. A screen rectangle of the given size is then reserved from `egui`, and each `DrawCall` is drawn to the screen recursively, depending on its `DrawCallKind`.

Note that the compound `DrawCallKind`s, like `Branch` or `Sequential`, do not draw to the screen themselves, but are only used for structuring the child `DrawCallId`s by moving them to their proper positions. For this purpose, the `RelativeDrawCallId` combines the `DrawCallId` of the child `DrawCall` with a pixel offset from the current parent `DrawCall`. Figure 5.5 illustrates how the entire function body is assembled from small primitive tokens, by color coding the background of each `DrawCall` with a unique color.

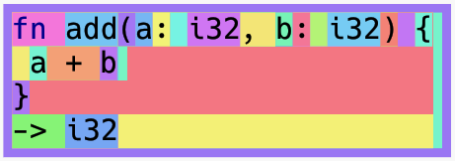


Figure 5.5: Render of a BIR body with each `DrawCall` rectangle colored with a distinct color.

Despite the introduction of this buffer leading to some dynamic memory allocations in the rendering code, it notably simplifies the rendering process of the BIR-tree. This was particularly considered a worthwhile trade-off during the initial design phase, as it increased the speed of development and iteration of different designs. No significant performance impact was observed with this algorithm, even when applied to large and complex function bodies. However, the potential for exploring more efficient rendering algorithms in the future remains.

5.4.2 User Interaction

During the draw buffer's construction, each `DrawCallId` is associated with the BIR node that inserted it into the draw buffer. This association serves two distinct purposes:

- It enables the mapping of screen-space user input back to BIR nodes.

- It facilitates the annotation of `DrawCall`s with the lifetime information computed during the analysis phase.

In the final rendering process, each `DrawCall` is assigned a screen rectangle by its parent `DrawCall`. Comparing these screen rectangles with the current cursor position allows for the identification of all `DrawCallId`s that the user might intend to interact with. However, even for simple functions, many `DrawCall`s will pass this basic 'contains'-check. For instance, the rectangle associated with the root `DrawCall` will always encompass any cursor position within the function body's bounds.

This complicates the determination of the exact element the user intends to interact with. Therefore, during the analysis, a subset of BIR nodes important to the ownership and reference analysis, or containing other relevant information, is extracted. Only `DrawCall`s associated with one of these predetermined nodes are considered when processing user input. If multiple options remain, the `DrawCall` nested deeper is selected, as it is typically more specific. While this approach has proven effective during testing, given the limited number of selectable nodes, a more advanced mechanism for resolving this ambiguity may be required for more complex cases in the future.

5.4.3 BIR Layouting

The tree-based structure of the BIR must be converted back to a more human-readable form for rendering. BORIS aims to output the BIR nodes in a format that closely resembles the original text-based Rust code. However, some modifications are necessary to properly annotate variable lifetimes, which will be discussed in the next Section 5.4.4.

```
1 let s = String::from("Hello");
2 if condition {
3     drop(s);
4 } else {
5     println!("{s}");
6 }
```

Listing 5.17: Example of variable `s` being conditionally dropped, but 'later' used in the `else`-branch.

Consider the code snippet in Listing 5.17. Here, `s` is conditionally moved/dropped in the `if`-branch at line 3. However, in the `else`-branch at line 5, it would still be considered alive, which would be problematic for the lifetime annotations, as the lifetime line would not be continuous.

This issue is not limited to branching code. It can occur whenever the execution order of an expression does not follow the top-to-bottom and left-to-right ordering of the source code text. For instance, the assignment operator (`=`) is evaluated from right-to-left. This issue is illustrated in Listing 5.18. Here, `x` is assigned in line 2, but the access to `x` in line 4 still refers to the initial value of `x` from line 1.

```
1 let mut x = 42;
2 x = {
3   println!("calculating..");
4   x + 1  // refers to the initial value of x
5 };
```

Listing 5.18: Non-branching code example where lexical ordering does not match the evaluation order.

To address these issues, BORIS renders code based on the evaluation order of the expressions. In the case of an assignment, this means that the right side of the assignment is drawn above the left side, as shown in Figure 5.7. For branching code, since there is no clear ordering between the evaluation of both branches (as they are executed mutually exclusively), they are rendered next to each other, as shown in Figure 5.6.

```
fn main() {
 let s = String::from("Hello");
 if condition {  else {
  drop(s);        println!("{s}");
 }               }
}
-> ()
```

Figure 5.6: BORIS output of Listing 5.17.

```
fn main() {
 let mut x = 42;
     {
      println!("calculating..");
      x + 1
     }
 x =
}
-> ()
```

Figure 5.7: BORIS output of Listing 5.18.

However, this layouting approach has been found to be rather unintuitive to developers, so it is only employed when necessary. For example, when the right side of an assignment is a simple expression containing only basic operators (e.g., `+`, `-`, `*`, `/`) and literals, edge cases like the one shown in Listing 5.18 are not possible. This means that for most cases, the left and right operands of an assignment can be drawn on the same 'line'. The sequential layouting (Figure 5.7) is used only if one of the operands contains more complex control flow, like a block- or a `match`-expression.

```
fn complex_arg(msg: &str) {
         add(if msg.starts_with("Hello") {  else {  )
              1                             2
             }                             }
             3
 let c =
 println!("{c}");
}
-> ()
```

Figure 5.8: Function call containing an argument with complex control flow rendered by BORIS.

The same principle is also used for the parameter list of a function call expression. By default, all function arguments can be drawn on the same line. However, if one of the arguments contains complex control flow, the argument expressions must be split into separate lines to ensure that lifetime annotations will behave properly in all edge cases. An example of such a complex function argument is shown in Figure 5.8.

All other BIR node kinds are handled in a similar manner. As there are many different kinds of BIR nodes, they are not all explained separately here. More information on how each specific node kind is handled can be found in the renderer's implementation.

5.4.4 Liveliness Annotations

In its default state, the application does not render any of the lifetime annotations, but only the function body itself. Only once a specific BIR node is selected by the user, the

liveliness information for this node is extracted from the BIR dependency graph (refer to Section 5.3.5) and incorporated into the final rendering. Showing the liveliness annotations for all variables simultaneously, has been found to clutter the visualization, making it harder to comprehend. Therefore, only the annotations for nodes relevant to the selection are shown. The traversal of the BIR dependency graph enables the identification of all nodes associated with the selected node. Currently, this includes the assignment and usage nodes of the selected variables, along with all nodes having a (potentially indirect) lifetime constraint to the selected variable.

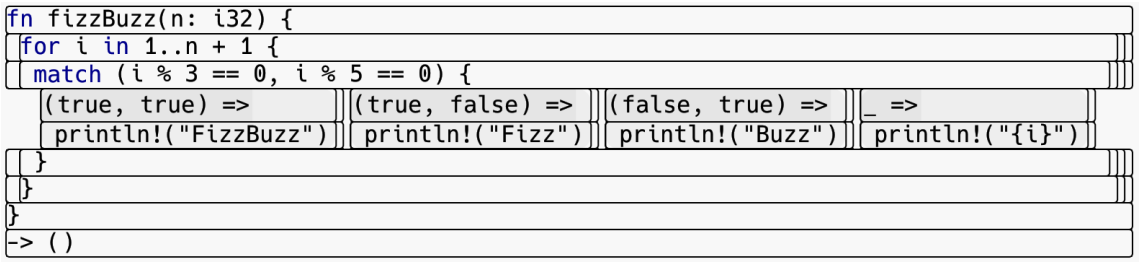


Figure 5.9: Debug view of the sequential sections extracted from the draw buffer.

Additional processing is performed during the transformation of the BIR body to the draw buffer (see Section 5.4.1) to render the liveliness annotations of all variables associated with the current selection. When a `Block`-expression, which represents a scope (`{}`) in the source code, is added to the draw buffer, all child `DrawCall`s, representing the blocks' statements, are traversed. Using the structure provided by the `Inline`-, `Sequential`-, and `Branch-DrawCallKind`s, the block is separated into a list of sequential sections, roughly representing source code lines. Figure 5.9 illustrates the outlines of these detected sections for a short source code example.

If a selected variable is owned by the current scope, a vertical indicator line is inserted on the left side of each section where the variable is determined to be 'live'. To differentiate between mutable and immutable variables, annotations for mutable variables or those containing a mutable reference are drawn with a thicker stroke. Different line styles could also be used for better differentiation between different kinds of variables, but altering the lines' thickness was the most straightforward implementation supported by `egui` during the initial design phase.

Sections containing a branch in control flow are handled slightly differently. In these cases, the liveliness lines are only split and merged at the entry and exit of the `Branch`-section, and the actual lifetime annotations are handled recursively by the scope associated with each branch.

When an active variable is assigned to or used in a current section, a horizontal line is drawn to the point of the usage/assignment in this section. The style of this line is slightly altered depending on the kind of usage to visually differentiate the different usage kinds.

- **Move**: The vertical line diverges to the right, indicating the movement of the value. A small triangle is drawn at the end of the move-line.
- **Copy**: No special indicator is drawn; the horizontal line simply branches off the vertical liveliness line, as a copy does not alter the original value.
- **Reference**: A small arrow is drawn at the intersection point, indicating that the reference 'points to' the variable.
- **Dereference**: A small circle is drawn at the intersection point to differentiate the dereference from a basic copy operation.

- **Assignment**: Assignments are represented similarly to a move, but in reverse direction as the value is moved into the variable.

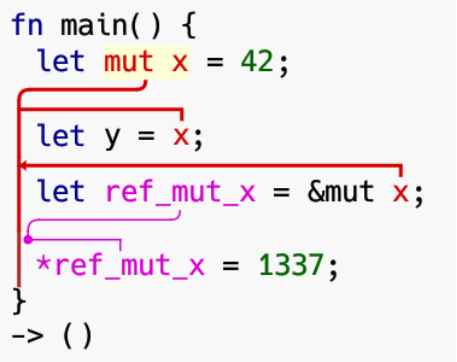


Figure 5.10: Basic example of a variable's usage annotations.

Figure 5.10 presents a simple example where a variable `x` is initialized in the first line. Since `x` is a basic numeric type, it implements the `Copy` trait, so its value copied to `y` in the subsequent line. A mutable reference to `x` is then created, as indicated by the small arrow pointing to `x`'s liveliness line. This reference is stored in `ref_mut_x`, and as the lifetime of `ref_mut_x` is constrained by `x`, its lifetime annotation is also shown in the visualization. In the final expression, this reference is dereferenced, assigning a new value to `x`.

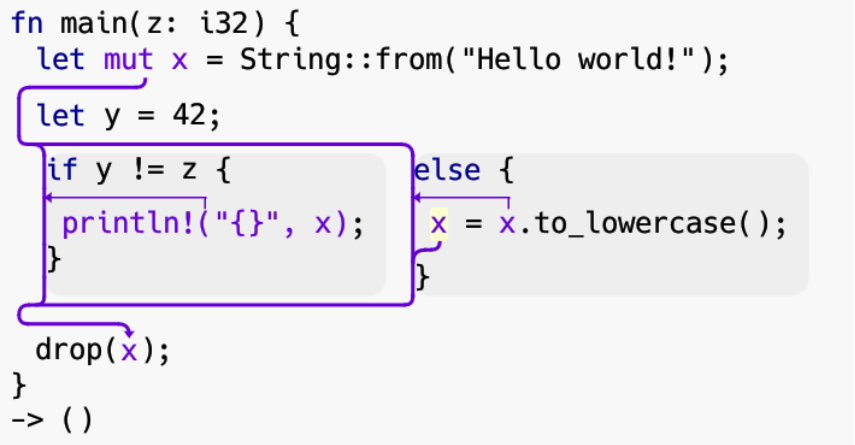


Figure 5.11: Example of a variable's lifetime being annotated within a branching code example.

Figure 5.11 illustrates a more complex example involving branching control flow. The branches, which are executed mutually exclusively, are depicted side-by-side. The lifetime line associated with `x` is split and rejoined after the `if`/`else`-statement. In the final statement of the `main`-function block, `x` is moved into the `drop`-function, as `String` does not implement the `Copy`-trait.

5.4.5 Conflict Rendering

During the traversal of the MIR, basic conflict detection is performed as outlined in Section 5.3.3. With the established mapping between BIR-nodes and `DrawCallIds`, these detected conflicts can be marked in the resulting render without requiring additional processing.

In the current implementation, conflict highlighting is achieved by drawing a red outline around the screen rectangle of a `DrawCall` that is associated with a node marked with a

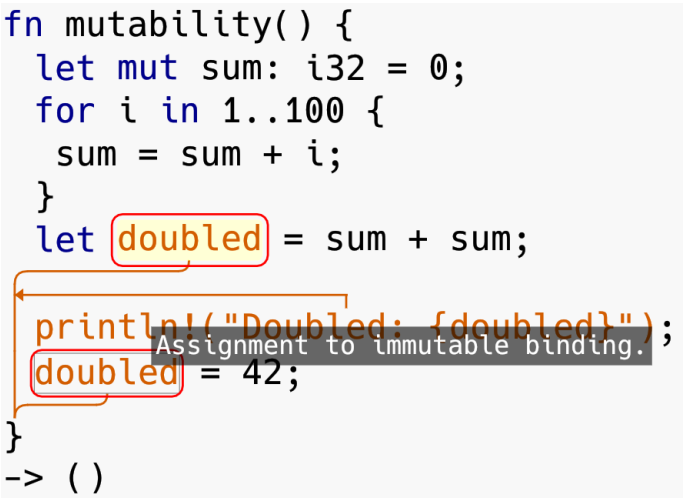


Figure 5.12: Basic rendering of conflict areas, with hover hint (cursor not shown).

conflict. Additionally, when hovering over the affected screen rectangle, a basic hover text is displayed to explain the error, as illustrated in Figure 5.12.

5.4.6 Resugaring

As elaborated in Section 4.1, the compiler desugars certain language constructs, such as `for`-loops and macros, during the lowering process from text-based source code to HIR, for simplifying the subsequent compilation and analysis processes. However, since the BIR is derived directly from the HIR, this desugaring also becomes evident in the final rendering of the BIR body. This discrepancy between the rendered function body and the original source code can lead to a confusing experience for developers, making the visualizations more challenging to interpret and thus less useful. To mitigate this issue, BORIS includes an additional processing step before generating the final visualization. This step identifies the parts that were desugared during the HIR-lowering process and provides the renderer with information on how to revert this desugaring.

5.4.6.1 `macro!` Expansion

Macros are a common feature used in Rust. Their use resembles a function calls but can be distinguished by a `!`-character following the macro's name. For instance, `println!("")` and `vec![]` are macro calls provided by Rust's standard library and are frequently used in Rust programs.

```
1  fn greet(name: &str) {
2      // original: format!("Hello {}!", name)
3      {
4          let res = $crate::fmt::format(
5              Arguments::new_v1_formatted(
6                  &["Hello␣", "!"],
7                  &[Argument::new_display(&name)],
8                  &[Placeholder::new( ... )],
9                  unsafe { UnsafeArg::new() }));
10         res
11     }
12 }
```

Listing 5.19: Slightly simplified expansion of the `format!`-macro generated from RA's HIR body.

Listing 5.19 illustrates how seemingly simple macro calls, such as `format!("Hello {}!", name)`, can expand into complex and difficult-to-read forms. To address this, BORIS identifies all root HIR-nodes originating from a macro call and associates them with the source code text of the macro call, e.g., `format!("Hello {}!", name)`.

A challenge arises when the macro is invoked with a variable from the function's scope, like `name` in Listing 5.19. Since the macro expansion is part of the AST generation, there are no HIR nodes generated corresponding to the macro call and its arguments. This makes it difficult to map a usage or assignment of the arguments within the macro expansion back to the argument in the macro call on the source code level. Furthermore, a macro may use and assign a variable passed via its arguments multiple times, making it impossible to annotate the variable usages in the re-sugared form of the macro call unambiguously.

BORIS addresses this issue by displaying the expanded form of the macro only if the expansion contains multiple active usages or is explicitly expanded, otherwise reverting to the contracted form. If an active variable is used in the expansion exactly once, the contracted form is displayed, and the entire macro-call-text is marked as a usage of that variable, as the exact position of the variable in the macro's arguments currently cannot be determined exactly. This approach was adopted following initial survey feedback (see Section 6.3.3) indicating that the expanded form of most macros is overly large and distracting.

5.4.6.2 `while`, `for`, and `?` Resugaring

Some common programming patterns in Rust have received some syntactic sugar, for enhancing readability, however during compilation, these constructs are desugared to their original form. For instance, the '`while EXPR { BODY }`'-loop desugars to a basic `loop`-expression containing a branch [62], as depicted in Listing 5.20.

```
loop {
    if EXPR {
        BODY
    } else {
        break
    }
}
```

Listing 5.20: Desugaring of a '`while EXPR { BODY }`'-loop.

Similarly, Rust's `for`-loop is syntactic sugar for the expansion shown in Listing 5.21, where the `while let`-loop is desugared as described above.

```
let mut iter = IntoIterator::into_iter(EXPR);
while let Some(PAT) = iter.next() {
    BODY
}
```

Listing 5.21: Desugaring of a '`for PAT in EXPR { BODY }`'-loop.

Another desugaring applied during the lowering process to HIR is the `?`-operator, which is used for Rust's error handling. It is syntactic sugar for an early return in case of an error, and unwraps the contained value otherwise [63]. The exact semantics of these operations are not crucial for this section.

BORIS traverses the HIR body of the function, searching for nodes originating from such a desugaring performed during HIR-lowering. It then records how the BIR nodes would have been arranged before the desugaring was performed. This allows the renderer to

```
fn question_mark() {
 let x: Result<i32, String> = may_fail();
        match branch(x)
        Continue(<ra@gennew>1) =>  Break(<ra@gennew>2) =>
        <ra@gennew>1               return from_residual(<ra@gennew>2)
 let y =
 println!("Success: {}", y);
 Result::Ok(())
}
-> Result<(), String>
```

Figure 5.13: Expanded rendering of the `?`-operator, keeping the `println!`-macro in a shorter contracted form.

toggle between the original and desugared representations while rendering these constructs. By default, all detected desugarings are rendered in their compact form for improved readability, but the user can switch to the desugared version manually, by clicking on the contracted version. Figure 5.13 displays the explicitly desugared version of the `x?` expression, while preserving the contracted form of the `println!(..)`-macro.

5.4.6.3 Parentheses Resugaring

As the tree based structure of the HIR implicitly encodes the operator associativity, parentheses are not required and are thus not encoded. However, when transforming the BIR back to a more textual representation, parentheses have to be reinserted.

Given for example a simple arithmetic expression:

```
1 a * (b + c)
```

In HIR this would be encoded as a multiply-expression on the top level, with an addition-expression as the right child. The parentheses around `(b + c)` are implicitly encoded in this parent-child-relationship of the HIR-expressions. However, when transforming these HIR-expressions back to a textual representation without further processing, it would result in

```
1 a * b + c
```

changing the result of the evaluation.

One possible solution would be adding parentheses around every child expression, resulting in:

```
1 (a) * ((b) + (c))
```

With this the semantic meaning is preserved correctly, but the resulting visualizations would become unnecessarily complex. Instead, BORIS only adds parenthesis around the child expression, if the expression precedence of the child-expression is lower than the parent-expression. The precedence of Rust expressions is defined in the documentation [64].

6 Evaluation

Quantitatively evaluating the effectiveness of learning and development tools, such as BORIS, presents a complex task. Initially, a quantitative metric for performance evaluation must be developed and measured to establish a baseline performance metric. Subsequently, the tool needs to be integrated into the participant's workflow, and this metric is reevaluated to ascertain any statistically significant performance improvement. In a teaching environment, this metric could be quiz performance, while in a development environment, it could be the time required for debugging borrow checker related errors.

The scale and time necessary for conducting such a quantitative analysis are considered beyond the scope of this thesis. However, in a similar work described in Section 3.1, the authors of *Aquascope* evaluated the impact of introducing their visualization tool, along with other interventions, for teaching new Rust developers. Their large-scale quantitative analysis observed a statistically significant increase in participants' performance on a set of quizzes [30]. Although this result does not directly provide insights into how BORIS might perform in a similar evaluation, it sets a precedent that such visualizations can positively impact learners' performance.

To evaluate the visualizations generated in this thesis, Section 6.1 discusses some cases where BORIS fails to generate valid results and potential fixes. Section 6.2 then qualitatively compares outputs from BORIS to results from previous works. Finally, a survey conducted with Rust developers of varying experience levels qualitatively evaluates the visualizations in Section 6.3.

6.1 Limitations

This section presents instances where the previously described analysis or visualization approach fails to produce valid outputs. These instances allow assessing the current state and applicability of the visualizations across different use-cases. Furthermore, these limitations can be used as guidance for future improvements.

6.1.1 Complex lifetime Annotations

As stated in Section 5.3.1, RA currently lacks support for lowering lifetime annotations. Consequently, BORIS is unable to handle these situations appropriately. Instead, it employs a conservative approximation in its analysis, which may lead to inaccurately extended

Figure 6.1: Conservative lifetime extension by the BORIS analysis.

reference lifetimes in certain cases. Figure 6.1 illustrates an instance where BORIS fails to correctly extend the lifetime annotations.

In this example, the `Container` struct, defined in Listing 6.1, has two fields, `a` and `b`, each referring to distinct reference lifetimes.

```
1 struct Container<'a, 'b> {
2     a: &'a str,
3     b: &'b str,
4 }
```

Listing 6.1: Struct with complex lifetime annotations used in Figure 6.1.

In this scenario, `rustc` could infer correctly that `a_ref` references `hello`, but not `world`. However, since BORIS cannot differentiate between the two lifetimes, it assumes that `container.a` also refers to `world`, resulting in an incorrect extension of the lifetime.

Such cases are rare in code snippets used for teaching Rust. However, in more complex scenarios, this can lead to minor inaccuracies in the BORIS analysis. Once RA implements proper lifetime annotation lowering, the reference tracking analysis of BORIS can be enhanced to match the `rustc` borrow checker's analysis accuracy.

6.1.2 Closure Captures

The issue depicted in the previous section becomes more apparent in the context of closures that capture their environment by reference. As outlined in Section 5.3.2, a closure is represented similarly to the struct depicted in Listing 6.1 internally. Each captured reference is stored as a separate field within the closure, each associated with a distinct lifetime. However, due to BORIS' inability to differentiate between these lifetimes, usages of the references within the closure bodies seem to refer to the same lifetime.

Moreover, the entire captured environment is passed as a single `self`-parameter to the closure body. This means that accesses to the captured variables are represented as projections of `self` at the MIR level. Consequently, all accesses to captured variables within the closure body appear to merge into a single 'lifetime'-line in the BORIS visualization, specifically, the line for the `self`-parameter of the closure.

This effect is visible in Figure 6.2, where it appears that `x` and `y` refer to the same entity within the closure body. Since this `self`-parameter of the closure is not represented at the HIR level and only exists after lowering to MIR, this behavior could potentially confuse developers who are not familiar with the intricacies of closure lowering in Rust. To address this issue, a specialized approach would be necessary in the analysis to map accesses of the closure environment to their corresponding variables in the captured scope.

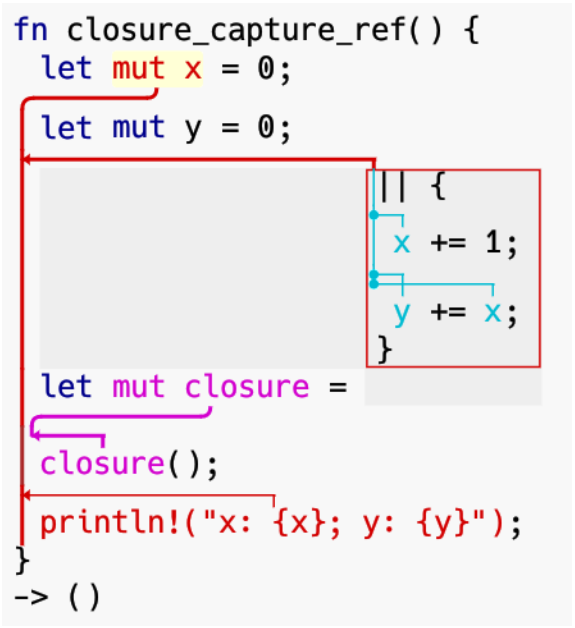


Figure 6.2: Example of a closure capturing multiple variables by reference.

6.1.3 Interior Mutability

When a variable is marked with the `mut`-keyword, the *exterior* mutability of the variable is set to mutable. As previously discussed, this allows the variable to be borrowed mutably (`&mut`), and its value to be modified after initialization.

Rust also provides another form of mutability, known as *interior* mutability, for situations where *exterior* mutability proves too restrictive [65]. The Rust standard library offers *interior* mutability through the `Cell<T>` and related types, such as `RefCell`. Fundamentally, a `Cell<T>` is just a wrapper type containing any value of the generic type `T`. The key function implemented on the `Cell<T>`-type is the `set`-method, which has the following signature:

```
impl<T> Cell<T> {
    pub fn set(&self, val: T) { .. }
    ..
}
```

As the name implies, this method sets the value of type `T` within the `Cell<T>` to `val`. Unexpectedly, this method takes an *immutable* reference to `self` instead of a *mutable* one. According to Rust's borrowing rules, it would be impossible to implement this function in a way that the borrow checker would accept, as `&self` may not be modified.

Consequently, `Cell::set()` has to be implemented with `unsafe`-code internally. *Interior* mutability allows for the modification of variables that appear immutable from the outside.

In a similar vein, `RefCell<T>` provides a `borrow_mut`-method with the following signature:

```
impl<T> RefCell<T> {
    pub fn borrow_mut(&self) -> RefMut<'_, T> { .. }
    ..
}
```

It allows for the acquisition of a *mutable* reference (encapsulated in the `RefMut<'_, T>` wrapper type) to the contained data while only requiring an *immutable* reference to `self`.

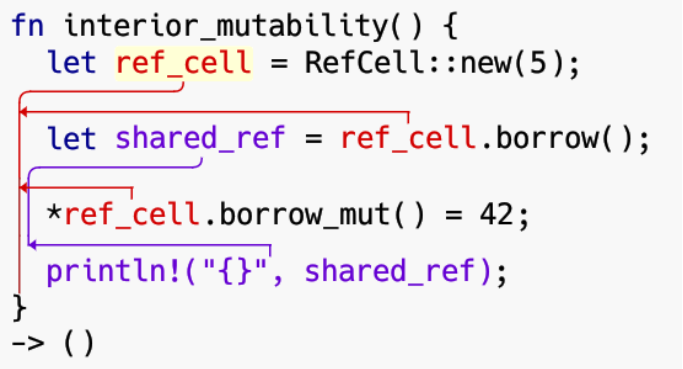


Figure 6.3: Interior mutability visualized by BORIS.

This added flexibility, however, is not without cost, as it shifts the borrow aliasing checks from compile time to runtime. With `RefCell`, the developer is responsible for adhering to the borrowing rules imposed by Rust. If these rules are violated, for instance, by attempting to borrow immutably while a mutable borrow is still active, the program will panic and exit immediately.

Figure 6.3 presents an example of interior mutability visualized by BORIS. As `ref_cell` is only borrowed immutably, this example appears valid to both BORIS and the Rust borrow checker. However, at runtime, the call to `ref_cell.borrow_mut()` would panic, as `shared_ref` holds an active reference to `ref_cell`.

By using interior mutability, or more specifically, `unsafe`-code in general, the developer explicitly opts out of the safety guarantees provided by the Rust borrow checker. This, in turn, limits the utility of tools based on Rust's borrowing rules, such as BORIS.

Other tools, such as MIRI [66], an interpreter for Rust's MIR, are being developed for finding vulnerabilities and undefined behavior in Rust programs that utilize `unsafe`-code [67]. Output generated by such tools could be used for extending visualizations containing such `unsafe`-code sections in the future.

6.1.4 `async` Code

Rust provides language-level support for writing asynchronous (`async`) code, enabling the creation of safe concurrent programs. This thesis does not delve into the exact syntax and semantics of `async` code, as `async` Rust can currently not be supported by BORIS.

The primary reason for this is that RA does not yet support the lowering of `async` code to MIR, rendering the previously described analysis unfeasible. Furthermore, without complex lifetime lowering support (refer to Section 6.1.1), the visualizations that BORIS would generate for `async` code would provide limited benefit. Owing to the manner in which `async` code is lowered by `rustc` to `Future`-trait objects [68], a problem akin to the closure visualization (see Section 6.1.2) would also arise with `async` code.

6.2 Comparisons

This section recreates some example results from previous works using BORIS to facilitate a comparison of the advantages and disadvantages of different approaches.

6.2.1 *Aquascope*

The updated ownership chapter of the Rust book [69] contains numerous visualization examples generated by *Aquascope*. Figure 6.4 presents one such example, which aims to visualize the issue of aliasing mutable and immutable references.

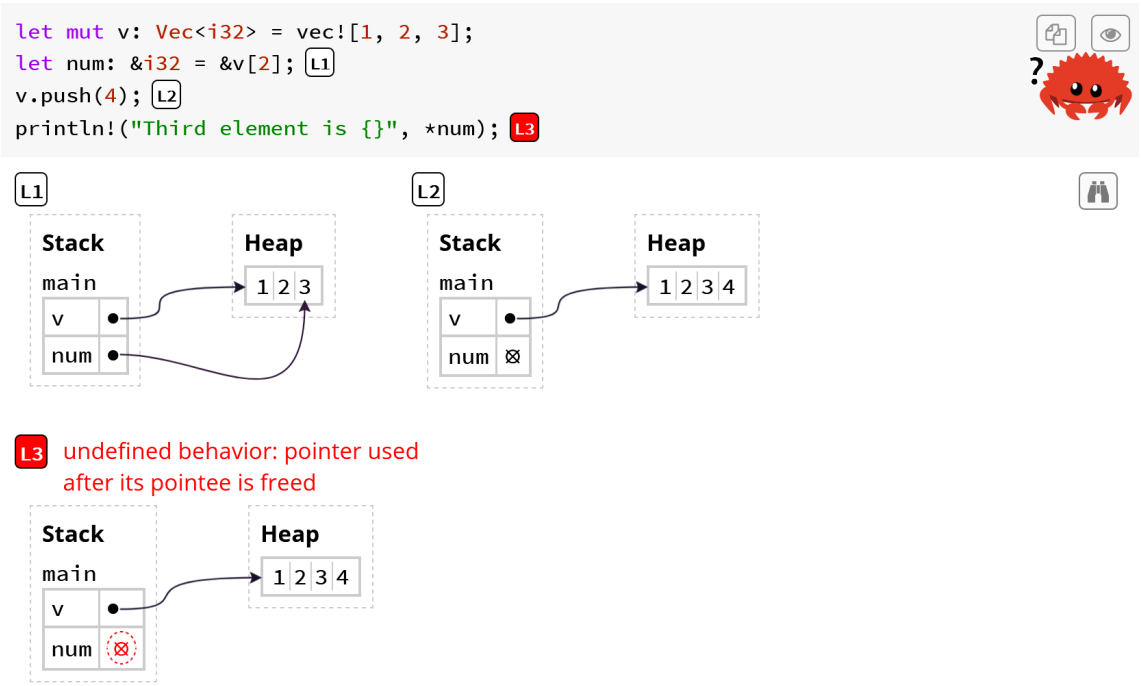


Figure 6.4: Example of aliasing references from the updated ownership chapter of the Rust book [69].

At location `L1`, `num` is shown to refer to heap memory owned by `v`. The call to `v.push(4)` at location `L2` implicitly borrows `v` mutably and may cause a reallocation of the owned heap memory if the vector's capacity is insufficient. This mutable borrow of `v` at location `L2` is illegal according to Rust's ownership rules, as it aliases with the immutable reference `num`, which remains active as it is used later at location `L3`. *Aquascope* illustrates why this reference aliasing would be problematic: at location `L3`, the reference `num` could point to invalid memory, in case the vector's heap memory was reallocated by the mutation at `L2`.

For comparison, Figure 6.5 displays the same example visualized by BORIS.

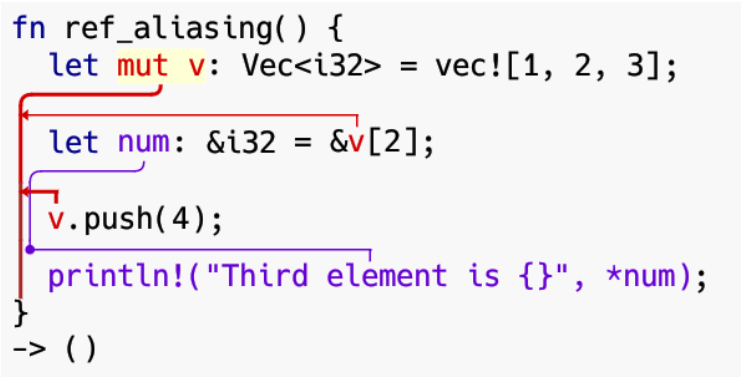


Figure 6.5: Example from Figure 6.4 visualized with BORIS.

While the BORIS visualization does not depict how the reference `num` could be invalid at location `L3`, it clearly demonstrates the aliasing of the references. This is evident from `v.push(4)` borrowing `v` while the liveliness line of `num` is still active.

As per the design objective of *Aquascope*, their visualization effectively illustrates the underlying memory error that Rust's borrow checker safeguards against. This can be instrumental in understanding the rationale behind the borrowing rules. However, this

depiction of the memory layout results in a relatively large footprint for the visualization, rendering it less suitable for larger scale examples. This is where the more compact visualization style of BORIS could offer an advantage. Both visualizations clearly pursue different objectives, making a direct comparison of the two approaches challenging.

6.2.2 *RustViz*

The most direct comparisons can be drawn between BORIS and *RustViz*, as both tools aim to visualize mostly the same concepts but with slightly different approaches. *RustViz* annotates the lifetime spans of variables/references adjacent to the source code, while BORIS integrates these annotations into the source text rendering. Both of these approaches have their own merits and drawbacks.

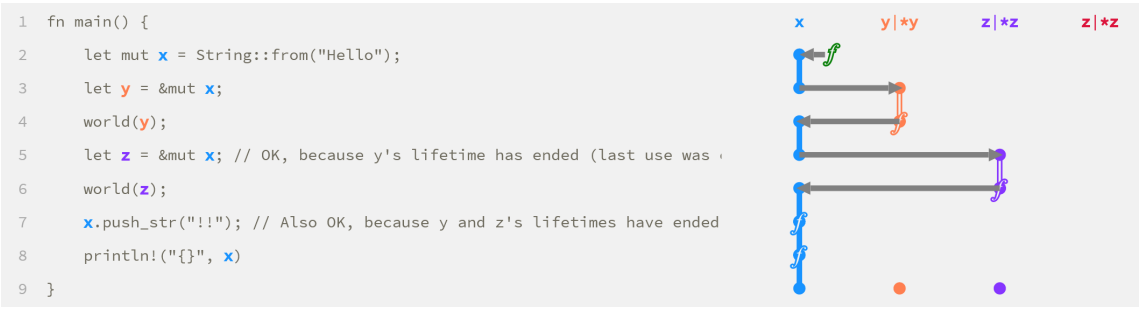


Figure 6.6: An example illustrating NLLs visualized by *RustViz* [35].

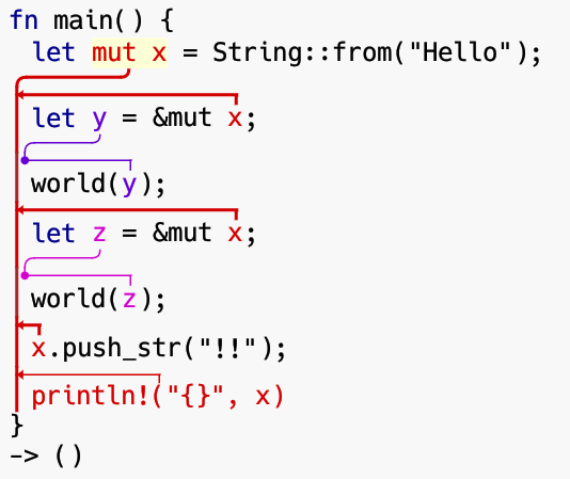


Figure 6.7: Example from Figure 6.6 visualized with BORIS.

RustViz, rendering the code as mostly plain text in a conventional layout, can be easily read by any developer. In comparison, visualizations generated by BORIS may necessitate some adaptation due to the automatic and unconventional source code layouting.

Conversely, *RustViz* currently lacks the capability to meaningfully visualize lifetimes in branching code. Moreover, all visualizations in *RustViz* are generated by manual annotation of the source code, whereas BORIS can automatically analyze Rust code.

Figures 6.6 and 6.7 compare how *RustViz* and BORIS annotate the same code example, illustrating the concept of NLL. The function `world` in both examples refers to:

```
1 fn world(s: &mut String) {
2     s.push_str(",␣world")
3 }
```

Both examples clearly demonstrate the effects of NLL, extending the lifetime of a reference up to its last usage. As noted in their own evaluation, *RustViz* requires more left-right eye movement for understanding the visualization, while BORIS presents the lifetime information in a more compact manner.

One difference between the visualizations is that BORIS omits any comments present in the original source code, as these are disregarded during lowering to HIR. This issue could be addressed by implementing an additional resugaring algorithm, as discussed in Section 5.4.6. However, it is unclear how well code comments would integrate with BORIS' custom layouting algorithm. As the comments are not represented in HIR, it is also unclear how their position would be mapped to HIR without modifications to the lowering code implemented by RA.

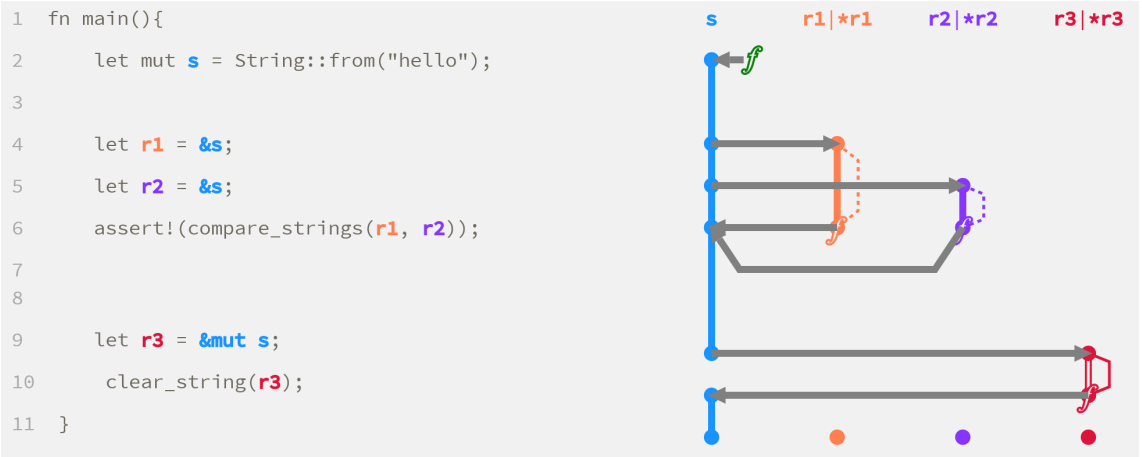


Figure 6.8: Basic example of Rust's borrowing mechanics annotated by *RustViz* [35].

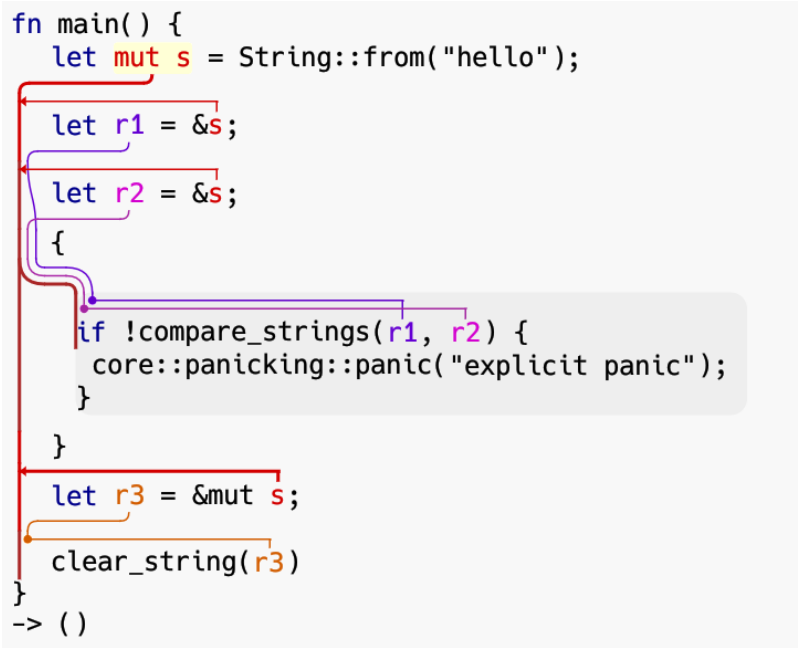


Figure 6.9: Example from Figure 6.8 visualized with BORIS.

Figures 6.8 and 6.9 present a comparison of a different example using these tools. Both visualizations again depict the same core information, so the preferred version will likely depend on the viewer.

One significant difference between the two visualizations is that BORIS displays the ex-

panded form of the `assert!`-macro. This is due to the presence of two active usages within the macro, which prevents BORIS from correctly annotating the contracted form (as described in Section 5.4.6.1). Consequently, BORIS defaults to showing the expanded macro. While this approach provides additional information in the visualization, it may also lead to confusion among users who are unfamiliar with the internal mechanics of macros.

6.2.3 *RustLifeAssistant*

```
1  fn main() {
2      let mut x = 4;
3      let y = &x;
4      let d = &x;
5      let y2 = move || {
6          println("{}", y);
7      };
8      let y3 = y2;
9      let e = &d;
10     let mut g = 5;
11     let z = bar(&y3);
12     let f = &mut g;
13     let w = foobar(&z);
14     let mut a = 32;
15     let b = 42;
16     let s = &w;
17     let r = s;
18     x = 5;
19     *f = 42;
20     take(g);
21     take(w);
22 }
23
24 fn foo<T>(p: T) -> T { p }
25 fn bar<T>(p: T) -> T { p }
26 fn foobar<T>(p: T) -> T { p }
27 fn take<T>(p: T) { }
```

Listing 6.2: Intentionally obfuscated lifetime error from the *RustLifeAssistant* evaluation [39].

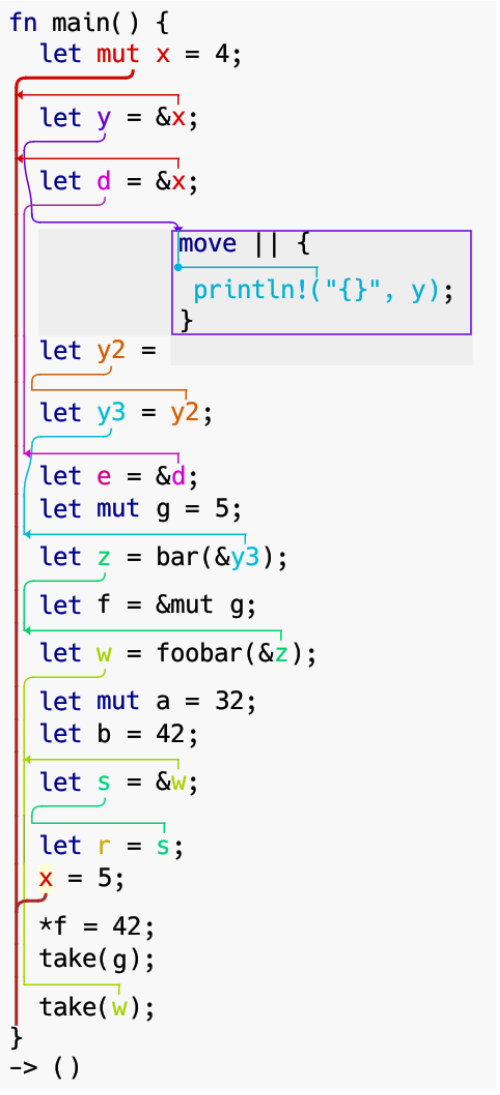


Figure 6.10: BORIS output of Listing 6.2.

In contrast to BORIS, *RustLifeAssistant* (refer to Section 3.5) is not designed to visualize the flow of ownership and lifetimes throughout the program. Instead, it aims to assist in debugging errors related to lifetimes. Although debugging such borrow checker errors was not a primary objective during the development of BORIS, it is still interesting to evaluate how tools like BORIS could aid in debugging.

Listing 6.2 presents a code example from the *RustLifeAssistant* thesis, which contains an intentionally obfuscated borrow checker error. The error arises in line 18, where `x` is reassigned while still being borrowed. However, the reason for the active borrow is concealed behind multiple layers of indirection and some unrelated code, so it is not obvious to the developer on first sight.

Rust compiler error (basically stderr of rustc):

```
error[E0506]: cannot assign to `x` because it is borrowed
  --> /media/david/Daten/Daten/ETH_Studium_Informatik/BSc_Thesis_Rus
   |
3  |     let y = &x;
   |             -- borrow of `x` occurs here
...
20 |     x = 5;
   |     ^^^^^ assignment to borrowed `x` occurs here
...
23 |     take(w);
   |          - borrow later used here

error: aborting due to previous error

For more information about this error, try `rustc --explain E0506`.
```

Possible explanation for "Why is w still borrowing the initial variable?"

1. "y" borrows the initial variable, due to line 3: 'let y = &x;'
2. "y2" borrows "y", due to line 5: 'let y2 = move || {'
3. "y3" borrows "y2", due to line 8: 'let y3 = y2;'
4. "z" borrows "y3", due to line 11: 'let z = bar(&y3);'
5. "w" borrows "z", due to line 13: 'let w = foobar(&z);'
6. "w" is later used

Figure 6.11: Output of the *RustLifeAssistant* IDE plugin run on the code in Listing 6.2 providing a possible explanation for the borrow checker error [39].

Figure 6.11 displays the output from the *RustLifeAssistant* IDE plugin, which explains the error in a step-by-step manner. In contrast, Figure 6.10 shows the analysis of the same code example using BORIS. From this visualization, it can be observed that at the point `x` is reassigned (`x = 5`), `w` is still alive, as it is later used by the `take(w)` call. Although it is not immediately clear why `w` contains a reference to `x`, the visualization can be utilized to trace `w` back to `z`, `y3`, `y2`, `y`, and finally `x`. Although the causal chain is not as explicitly illustrated as in the output from *RustLifeAssistant*, BORIS effectively communicates the same information through its visualization.

6.2.4 REVIS

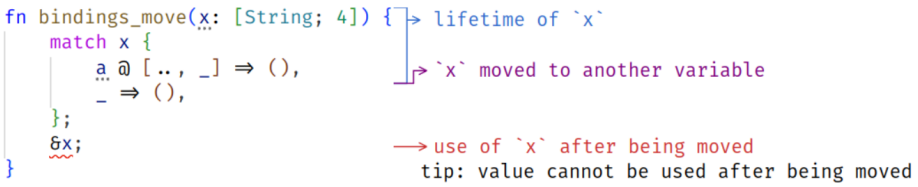


Figure 6.12: Code example containing a borrow checker error, annotated by REVIS.

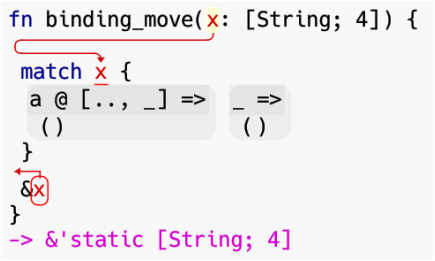


Figure 6.13: Example from Figure 6.12 visualized with BORIS.

REVIS, like *RustLifeAssistant*, aims to simplify the debugging of compiler errors (see Section 3.6). Unlike *RustLifeAssistant*, REVIS accomplishes this by interpreting the com-

piler's error output and converting it into visual annotations within the source code inside the IDE.

Figure 6.13 presents an example of REVIS's output, which visualizes a 'use after move'-error. For comparison, the same code example analyzed by BORIS is shown on the right. The variable `x` is clearly moved in the first line, rendering subsequent access invalid. While BORIS was not explicitly designed to detect and visualize such errors, it can still be useful for finding and debugging them.

6.3 Survey Evaluation

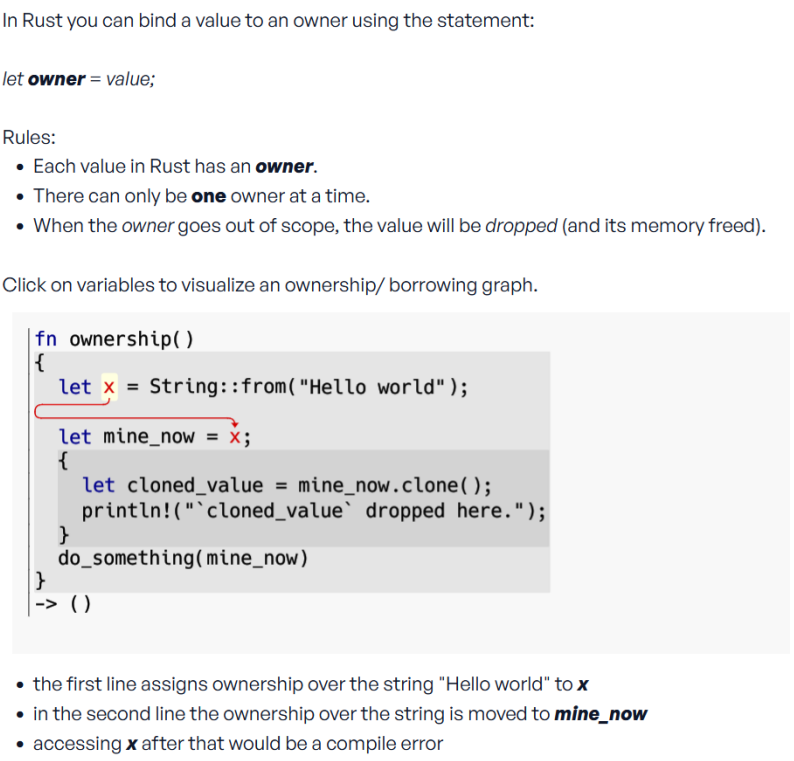


Figure 6.14: Excerpt from the survey page, explaining Rust's core ownership principles.

To evaluate the visualizations generated in this thesis in a more formal manner, a survey was conducted among Rust users of varying experience levels. Recognizing that some participants might not be familiar with Rust, the survey included brief explanations of Rust's fundamental ownership and borrowing mechanisms. These explanations were supplemented with visualizations of short code snippets, which were generated by BORIS to illustrate the concepts being explained. After engaging with these explanations and visualizations, participants were asked to qualitatively rate the visualizations on a 5-point Likert scale [70].

One of the primary considerations during the survey design was to ensure a high response rate. This was achieved by keeping the survey concise and the participation process straightforward. Ultimately, the survey comprised three questions to assess the participants' experience level and four questions to evaluate the visualizations. To avoid the significant obstacle of downloading the full standalone BORIS program, the visualizations were directly embedded into the survey website.

For this purpose, a separate viewer application was developed, which does not contain any of the analysis logic itself, but can only render pre-analyzed function bodies. The viewer

application was then compiled to WebAssembly (WASM) [71] to facilitate direct embedding into the survey webpage. Figure 6.14 presents the explanation of Rust's ownership semantics as depicted in the survey. A full version of this survey can be found in appendix A.1.

6.3.1 Participants

Participants were recruited by sharing the survey link on two different Rust forums: the official Rust user forum [72] and the `r/rust` subreddit [73]. For reaching a more diverse group of developers with different backgrounds, the survey was also shared on the `r/learnprogramming` reddit forum [74] and with the *Advanced Programming* course (winter term 2023/2024) at Julius-Maximilians University Würzburg, which included a brief introductory lecture on Rust.

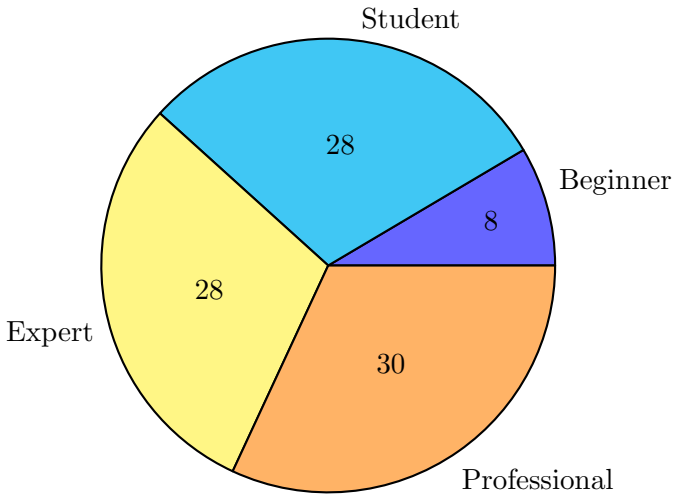


Figure 6.15: Distribution of self-assessed programming experience among survey participants.

Given the complete anonymization of the survey, it is impossible to trace specific survey submissions back to their sources. However, an approximate distribution can be inferred based on the timing of the forum posts, corresponding survey submission spikes, and page views from the forum posts. Most participants ($\sim 80\%$) originated from the `r/rust` post, while the remaining submissions were approximately equally divided between the Rust user forum, the `r/learnprogramming` post and the *Advanced Programming* course. Ignoring 4 incomplete submissions, the survey accumulated a total of 94 submissions, representing an $\sim 7.8\%$ submission rate relative to the 1203 survey page views.

In the first part of the survey, participants were asked to self-assess their general programming experience on a scale comprising:

- **Beginner**: Basic programming knowledge.
- **Student**: Actively learning.
- **Professional**: Over a year of industry experience.
- **Expert**: Over five years of industry experience.

Figure 6.15 illustrates the distribution of participants' self-assessed overall experience.

Participants were also asked to evaluate their experience with the Rust language. The results are depicted in Figure 6.16.

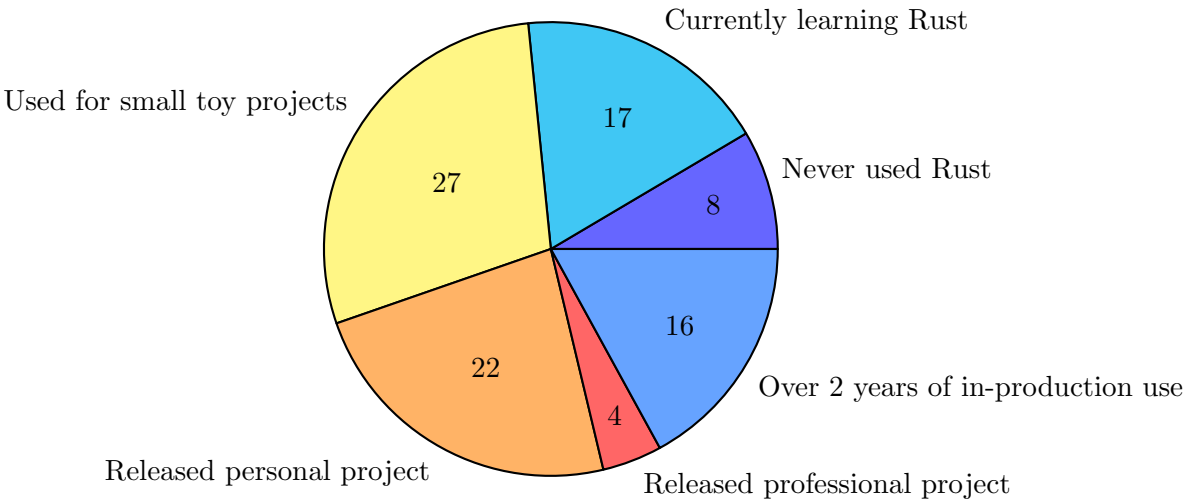


Figure 6.16: Distribution of self-assessed Rust experience among survey participants.

Given the relatively small size of the *Released professional project* group, it was merged with the *Over 2 years of in-production use* group into a broader *Expert* category for all further analysis in Section 6.3.2. Overall, the survey reached a fairly even distribution between beginner, intermediate, and professional participants.

6.3.2 Results

At the end of the survey, participants were asked to rate four statements about the visualizations on a five-point scale:

1. **They are intuitive to use and grasp** Would you detect errors faster compared to console-based borrow checker errors?
2. **They helped you to understand Ownership and Borrowing** or deepened your (intuitive) understanding if you were already familiar with these concepts.
3. **They could help explain these concepts to beginners** Making Rust easier and faster to learn for new programmers, and ones switching from other languages.
4. **You would use such Visualizations during development** Assuming they were directly integrated into your IDE.

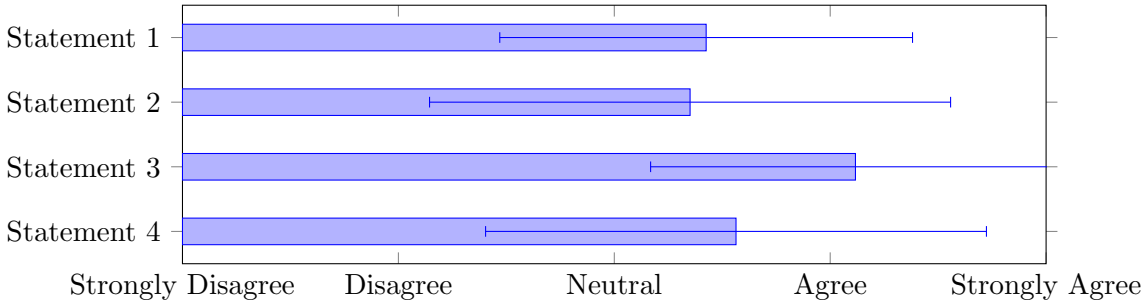


Figure 6.17: Average ratings of the 4 statements on a 5-point Likert scale [70].

On average, all statements received a positive rating between 3 (Neutral) and 4 (Agree). The statement *"The visualizations could help explain these concepts to beginners"* (Statement 3) received the highest average rating of ~ 4.12, indicating that most participants agree that the visualizations could be beneficial for beginners. However, the ratings for all

statements showed a relatively high standard deviation of about one point on the rating scale, suggesting a high variability in the perceived helpfulness of the visualizations among different participants.

The statement with the highest variability between submissions was *"The visualizations helped you to understand Ownership and Borrowing"* (Statement 2). As shown in Figure 6.18, this variability can be partially attributed to the varying levels of Rust experience among survey participants. Participants who have never used Rust or are currently learning it generally agreed with the statement, while more experienced Rust developers rated the statement as ~ 3 (Neutral) on the rating scale.

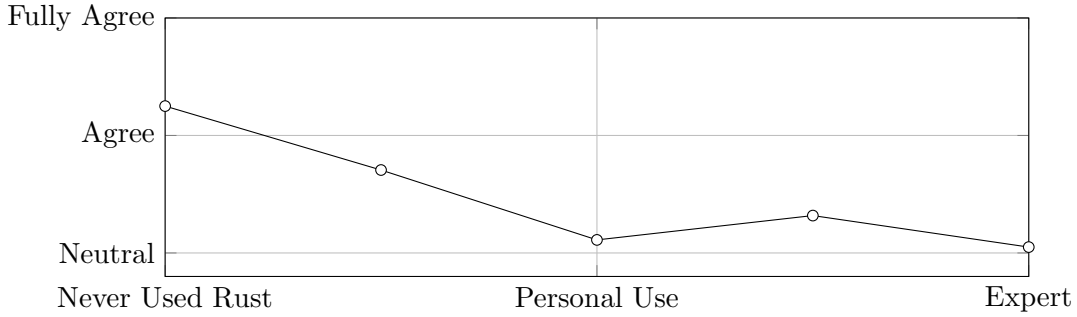


Figure 6.18: Average rating of the statement *"The visualizations helped you to understand Ownership and Borrowing"* (Statement 2) based on the participants' Rust experience level.

The survey results indicate that developers new to Rust tend to find the visualizations helpful. However, this trend suggests that the usefulness of the visualizations may plateau once a basic understanding of these concepts is acquired.

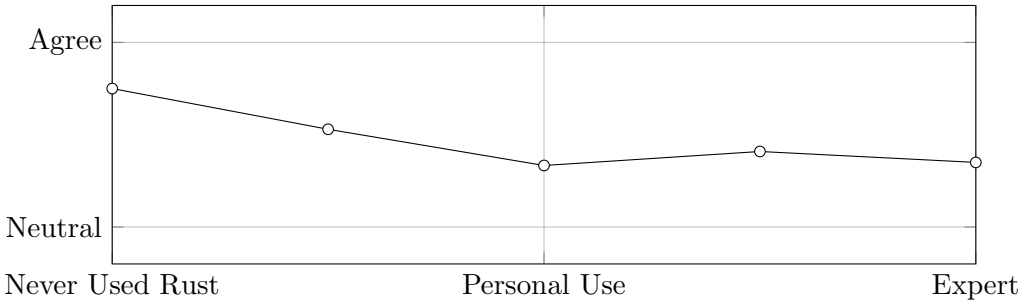


Figure 6.19: Average rating of the statement *"The visualizations are intuitive to use and understand"* (Statement 1) based on the participants' Rust experience level.

Another interesting, yet less significant, trend was observed in the average rating of the statement *"The visualizations are intuitive to use and understand"* (Statement 1), which inversely correlated with the Rust experience of participants, as shown in Figure 6.19. This may be due to more experienced Rust developers having already developed their own intuition of how they think about ownership and borrowing, making it more challenging for them to adapt to the visualizations.

Statement 3, *"The visualizations could help explain these concepts to beginners"*, was the only statement that exhibited negligible variation in relation to the participants' experience with Rust. This outcome is promising as it suggests that BORIS is broadly perceived as a beneficial tool in facilitating the learning process of Rust.

Statement 4, *"You would use such Visualizations during development"*, varied based on the participants' Rust experience, as shown in Figure 6.20. In general, it appears that new

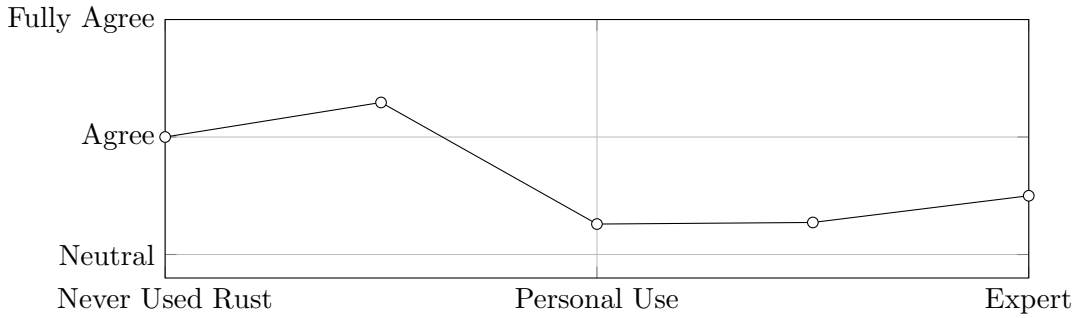


Figure 6.20: Average rating of the statement *"You would use such Visualizations during development"* (Statement 4) based on the participants' Rust experience level.

Rust developers are more inclined to integrate such visualizations into their development environment compared to more experienced developers. Possibly due to a similar reason as proposed for Statement 1, as experienced developers have already created their own development environment they are comfortable with, so it becomes harder to integrate new tools.

6.3.3 Comments

The survey concluded with an optional text entry field for participants to provide general feedback and comments. Several common points of feedback about the visualizations emerged from these comments:

- The `macro!`-expansion, particularly the `println!` macro expansion, was frequently mentioned as being too large and distracting from the core information. Here it has to be mentioned that at the time of the survey's creation, BORIS displayed the full macro expansion whenever a single variable within the macro expansion was active. Participants suggested displaying a simpler, less accurate representation by default and making the full `macro!`-expansion optional. This modification has since been incorporated into BORIS, significantly simplifying many macro-containing cases.
- Participants proposed improving the visual differentiation between moves, copies, and (de-)reference operations, possibly by adding small symbols to represent each operation.
- Participants also suggested using different types of lines to represent mutable and immutable values and references, for example, dashed versus solid lines.
- Some participants expressed concerns about potential scaling issues, which could make very long functions difficult to comprehend. Especially, for functions containing a lot of branching, as these cause the visualizations to become very wide.

Scaling visualizations to accommodate lengthy functions is a common, hard to solve limitation across various visualization approaches. However, BORIS could address most of the other concerns raised by participants in future updates. Notably, the issue of using distinct styles to differentiate between mutable and immutable variables primarily stems from the limited out-of-the-box support provided by the `egui` rendering framework and time constraints during the initial development of the application.

6.3.4 Threats to Validity

While qualitative analysis can reveal general trends in the perceived utility of the application, the survey's expressiveness is not without limitations. Although nearly 100

Name	Total Members	Post Views	#Submissions[1]	Rate[2]
`r/rust`	283.599	26.000	~ 75	$\sim 0.29\%$
`r/learnprogramming`	4.100.000	8.700	~ 5	$\sim 0.057\%$

Table 6.1: Comparison of the impact of the posts in the `r/rust` and `r/learnprogramming` forums.

participants suffice for identifying some general trends in the survey results, this sample size remains insufficient for drawing definitive conclusions.

Furthermore, a selection bias likely exists among the survey participants. Most participants were recruited from Rust-related forums, and those who volunteered for the study likely had a pre-existing interest in Rust. As a result, they may not represent a typical cross-section of developers.

This bias becomes evident when examining the approximate survey submission conversion rates from `r/rust` [73] compared to `r/learnprogramming` [74] posts (see table 6.1). Despite `r/learnprogramming` having over 14 times more members compared to `r/rust`, the post in `r/rust` received almost three times the number of views. Additionally, the approximate conversion rate from post views to survey submissions was about five times higher in `r/rust`. While various factors contribute to a post's impact in an online forum, this difference clearly suggests that developers with a prior interest in Rust were more inclined to participate in the survey.

Another potential threat to the survey's validity is the lack of participant verification, which means that a single individual could potentially have participated multiple times. This approach was adopted to simplify the participation process and to avoid conflicting with participant's anonymity. However, this issue is unlikely to significantly impact the results, as no malicious patterns were observed in the data, and participants had little incentive to submit multiple responses given that they stood to gain nothing from doing so.

[1] Approximate number of submissions originating from the post, based on submission spikes after each post. The `r/learnprogramming` post was made when traffic to the `r/rust` post was almost at 0.
[2] Approximate conversion rate from forum post views to survey submissions.

7 Conclusion

Rust's image of being a hard to learn programming language with a steep learning curve is partially responsible for its slowly increasing adoption. The borrow checker, in particular, can be daunting for new Rust developers who are yet to grasp Rust's ownership and borrowing rules. Previous works have demonstrated that tools and visualizations explaining Rust's ownership mechanics can facilitate the learning process [30], lowering the barrier of adoption for new developers.

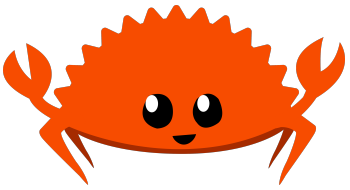
This thesis presented a novel approach to visualize how Rust's borrow checker interprets code. These visualizations are automatically generated through a custom analysis built on the official Rust LSP server, RA. Unlike previous works, BORIS can handle larger, more complex source code examples with complex, branching control flow. This is achieved by arranging the source code according to the execution order of expressions. The extracted ownership and reference lifetime information is directly annotated within this custom Rust code visualization. Allowing developers to explore dependencies and data flow between variables interactively.

A qualitative survey was conducted on developers with varying Rust experience levels to evaluate the outputs. The survey participants generally found the visualizations generated by this thesis to be usable and useful. The survey results suggest that these visualizations could be particularly beneficial for explaining Rust's ownership and borrowing mechanisms to new Rust developers, with a more limited application as a development assistance tool.

In their current state, the visualizations do still leave room for future improvements, making them more useful during general development. Key areas for future work include aligning the custom analysis implementation with current borrow checker implementation in the Rust compiler, which would necessitate some modifications to RA. Furthermore, integrating these visualizations into the development workflow to make them accessible from the IDE directly would make them more useful during development. Lastly, addressing the feedback provided by survey participants (as detailed in Section 6.3.3), especially about the visual design of the annotations, would make the outputs easier for developers to understand.

This thesis has introduced a novel approach for laying out Rust source code based on its evaluation order, making it more suitable for the annotation of lifetime information. The ownership and reference tracking analysis built directly on top of RA, facilitates integrating interactive BORIS visualizations directly into the IDE of developers in the future. Visual annotations of Rust's ownership and borrowing mechanics can potentially elevate some

of the cognitive overhead required for writing Rust, especially for developers new to the language.



List of Figures

Listings

Acronyms

CPU Central Processing Unit

RAM Random Access Memory

OS Operating System

GC Garbage Collector

IR Intermediate Representation

AST Abstract Syntax Tree

HIR High-Level Intermediate Representation

REVIS Rust Error Visualizer

MIR Mid-level Intermediate Representation

BORIS BORrow vISualizer

BIR BORIS Intermediate Representation

IDE Integrated Development Environment

RA `rust-analyzer`

RLS Rust Language Server

LSP Language Server Protocol

`rustc` Rust compiler

API Application Programming Interface

UI User Interface

RAII Resource Acquisition Is Initialization

NLL Non-Lexical Lifetimes

CFG Control Flow Graph

WASM WebAssembly

`async` asynchronous

Bibliography

[1] Stack Overflow, "2023 developer survey - most popular technologies." https://survey.stackoverflow.co/2023/#most-popular-technologies-language, 2024. Accessed: 2024-01-31.

[2] M. C. Sánchez, J. M. C. de Gea, J. L. Fernández-Alemán, J. Garcerán, and A. Toval, "Software vulnerabilities overview: A descriptive study," *Tsinghua Science and Technology*, vol. 25, no. 2, pp. 270–280, 2019.

[3] G. Thomas, "A proactive approach to more secure code." https://msrc.microsoft.com/blog/2019/07/a-proactive-approach-to-more-secure-code/, 2019. Accessed: 2024-02-02.

[4] J. V. Stoep, "Memory safe languages in android 13." https://security.googleblog.com/2022/12/memory-safe-languages-in-android-13.html, 2022. Accessed: 2024-02-02.

[5] Rust Foundation, "Rust - a language empowering everyone to build reliable and efficient software.." https://www.rust-lang.org/, 2024. Accessed: 2024-01-31.

[6] 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.

[7] Rust Foundation, "The rust reference - influences." https://doc.rust-lang.org/reference/influences.html, 2024. Accessed: 2024-01-31.

[8] Stack Overflow, "2023 developer survey - admired and desired." https://survey.stackoverflow.co/2023/#technology-admired-and-desired, 2024. Accessed: 2024-01-31.

[9] Rust Foundation, "Announcing rust 1.0." https://blog.rust-lang.org/2015/05/15/Rust-1.0.html, 2015. Accessed: 2024-01-31.

[10] Rust Foundation, "2022 annual rust survey results." https://blog.rust-lang.org/2023/08/07/Rust-Survey-2023-Results.html, 2023. Accessed: 2024-02-16.

[11] K. Cook, "Rust introduction for v6.1-rc1." https://lore.kernel.org/lkml/202210010816.1317F2C@keescook/, 2023. Accessed: 2024-03-06.

[12] Microsoft, "windows-drivers-rs." https://github.com/microsoft/windows-drivers-rs, 2024. Accessed: 2024-03-06.

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

[14] 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)*, pp. 597–616, 2021.

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

[16] Rust Foundation, "Rust survey 2020 results." `https://blog.rust-lang.org/2020/12/16/rust-survey-2020.html`, 2023. Accessed: 2024-02-16.

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

[18] L. Ferres, "Memory management in c: The heap and the stack," *Universidad de Concepcion*, 2010.

[19] C. Cowan, F. Wagle, C. Pu, S. Beattie, and J. Walpole, "Buffer overflows: Attacks and defenses for the vulnerability of the decade," in *Proceedings DARPA Information Survivability Conference and Exposition. DISCEX'00*, vol. 2, pp. 119–129, IEEE, 2000.

[20] K. Yang, Z. Huang, and M. Yang, "Array bounds check elimination for java based on sparse representation," in *2009 Seventh ACIS International Conference on Software Engineering Research, Management and Applications*, pp. 189–196, IEEE, 2009.

[21] K. Sen, "Race directed random testing of concurrent programs," in *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 11–21, 2008.

[22] CPP Reference, "Raii." `https://en.cppreference.com/w/cpp/language/raii`, 2024. Accessed: 2024-02-01.

[23] Rust Foundation, "The rust programming language - understanding ownership." `https://doc.rust-lang.org/book/ch04-00-understanding-ownership.html`, 2024. Accessed: 2024-02-01.

[24] V. Ragunathan and V. Ragunathan, "Nullptr," *C++/CLI Primer: For. NET Development*, pp. 49–50, 2016.

[25] Rust Foundation, "Module std::option." `https://doc.rust-lang.org/std/option/`, 2024. Accessed: 2024-02-29.

[26] Rust Foundation, "The rust rfc book - rfc 2094 nll." `https://rust-lang.github.io/rfcs/2094-nll.html`, 2017. Accessed: 2024-04-01.

[27] Rust Foundation, "The rust programming language - validating references with lifetimes." `https://doc.rust-lang.org/book/ch10-03-lifetime-syntax.html#validating-references-with-lifetimes`, 2024. Accessed: 2024-02-29.

[28] Rust Foundation, "The rustonomicon." `https://doc.rust-lang.org/nomicon/`, 2024. Accessed: 2024-02-01.

[29] V. Astrauskas, C. Matheja, F. Poli, P. Müller, and A. J. Summers, "How do programmers use unsafe rust?," *Proceedings of the ACM on Programming Languages*, vol. 4, no. OOPSLA, pp. 1–27, 2020.

[30] W. Crichton, G. Gray, and S. Krishnamurthi, "A grounded conceptual model for ownership types in rust," *Proceedings of the ACM on Programming Languages*, vol. 7, no. OOPSLA2, pp. 1224–1252, 2023.

[31] "Cognitive engineering lab." `https://github.com/cognitive-engineering-lab`, 2024. Accessed: 2024-02-15.

[32] Cognitive Engineering Lab, "The rust programming language - what is ownership?." `https://rust-book.cs.brown.edu/ch04-01-what-is-ownership.html`, 2024. Accessed: 2024-02-16.

[33] "Cognitive engineering lab - aquascope playground." `https://cognitive-engineering-lab.github.io/aquascope/`, 2024. Accessed: 2024-02-16.

[34] W. Crichton, M. Patrignani, M. Agrawala, and P. Hanrahan, "Modular information flow through ownership," in *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pp. 1–14, 2022.

[35] M. Almeida, G. Cole, K. Du, G. Luo, S. Pan, Y. Pan, K. Qiu, V. Reddy, H. Zhang, Y. Zhu, *et al.*, "Rustviz: Interactively visualizing ownership and borrowing," in *2022 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pp. 1–10, IEEE, 2022.

[36] P. Ruffwind, "Graphical depiction of ownership and borrowing in rust." `https://rufflewind.com/2017-02-15/rust-move-copy-borrow`, 2017. Accessed: 2024-02-15.

[37] J. Walker, "Rust lifetime visualization ideas." `https://blog.adamant-lang.org/2019/rust-lifetime-visualization-ideas/`, 2019. Accessed: 2024-02-16.

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

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

[40] R. Wang, M. Maclaren, and M. Coblenz, "Revis: An error visualization tool for rust," *arXiv preprint arXiv:2309.06640*, 2023.

[41] Rust Foundation, "The rust reference - whitespace." `https://doc.rust-lang.org/stable/reference/whitespace.html`, 2024. Accessed: 2024-02-01.

[42] Rust Foundation, "rustfmt." `https://github.com/rust-lang/rustfmt`, 2024. Accessed: 2024-02-01.

[43] Rust Foundation, "Rust compiler development guide - source code representation." `https://rustc-dev-guide.rust-lang.org/part-3-intro.html`, 2024. Accessed: 2024-02-01.

[44] Rust Foundation, "Rust compiler development guide - code generation." `https://rustc-dev-guide.rust-lang.org/backend/codegen.html`, 2024. Accessed: 2024-02-01.

[45] N. Matsakis, "Introducing mir." `https://blog.rust-lang.org/2016/04/19/MIR.html`, 2016. Accessed: 2024-02-01.

[46] Cognitive Engineering Lab, "rustc_plugin." `https://github.com/cognitive-engineering-lab/rustc_plugin?tab=readme-ov-file#installation`, 2024. Accessed: 2024-02-01.

[47] Rust Foundation, "rust-analyzer." `https://github.com/rust-lang/rust-analyzer`, 2024. Accessed: 2024-02-01.

[48] Microsoft, "Language server protocol." `https://microsoft.github.io/language-server-protocol/`, 2024. Accessed: 2024-04-18.

[49] Rust Foundation, "The rust rfc book - rfc 2912 rust-analyzer." `https://rust-lang.github.io/rfcs/2912-rust-analyzer.html`, 2020. Accessed: 2024-02-01.

[50] Rust Foundation, "rust-analyzer flycheck." `https://github.com/rust-lang/rust-analyzer/tree/master/crates/flycheck`, 2024. Accessed: 2024-02-01.

[51] Rust Foundation, "Enum rustc_middle::mir::syntax." https://doc.rust-lang.org/nightly/nightly-rustc/rustc_middle/mir/syntax/index.html, 2024. Accessed: 2024-02-07.

[52] Rust Foundation, "Rust compiler development guide - move paths." https://rustc-dev-guide.rust-lang.org/borrow_check/moves_and_initialization/move_paths.html, 2024. Accessed: 2024-02-07.

[53] Rust Foundation, "Function std::mem::replace." https://doc.rust-lang.org/std/mem/fn.replace.html, 2024. Accessed: 2024-02-08.

[54] A. B. Kahn, "Topological sorting of large networks," *Communications of the ACM*, vol. 5, no. 11, pp. 558–562, 1962.

[55] Rust Foundation, "Rust compiler development guide - mir borrow check." https://rustc-dev-guide.rust-lang.org/borrow_check.html, 2024. Accessed: 2024-02-08.

[56] Rust Foundation, "rust-analyzer - issue: Flowistry like feature." https://github.com/rust-lang/rust-analyzer/issues/14668, 2024. Accessed: 2024-02-24.

[57] Rust Foundation, "The rust programming language - closures." https://doc.rust-lang.org/book/ch13-01-closures.html, 2024. Accessed: 2024-03-01.

[58] Ferrilab, "Crate bitvec." https://docs.rs/bitvec/latest/bitvec/, 2024. Accessed: 2024-03-10.

[59] T. Busjahn, R. Bednarik, A. Begel, M. Crosby, J. H. Paterson, C. Schulte, B. Sharif, and S. Tamm, "Eye movements in code reading: Relaxing the linear order," in *2015 IEEE 23rd International Conference on Program Comprehension*, pp. 255–265, IEEE, 2015.

[60] E. Ernerfeldt, "egui." https://github.com/emilk/egui, 2024. Accessed: 2024-02-11.

[61] E. Ernerfeldt, "egui - disadvantages of immediate mode." https://github.com/emilk/egui?tab=readme-ov-file#layout, 2024. Accessed: 2024-02-11.

[62] Rust Foundation, "The rust rfc book - rfc 0214 while let." https://rust-lang.github.io/rfcs/0214-while-let.html, 2020. Accessed: 2024-02-14.

[63] Rust Foundation, "The rust rfc book - rfc 1859 try trait." https://rust-lang.github.io/rfcs/1859-try-trait.html, 2020. Accessed: 2024-02-14.

[64] Rust Foundation, "The rust reference - expression precedence." https://doc.rust-lang.org/reference/expressions.html#expression-precedence, 2024. Accessed: 2024-04-019.

[65] Rust Foundation, "The rust programming language - RefCell<T> and the interior mutability pattern." https://doc.rust-lang.org/book/ch15-05-interior-mutability.html, 2024. Accessed: 2024-02-28.

[66] Rust Foundation, "Miri." https://github.com/rust-lang/miri, 2024. Accessed: 2024-03-13.

[67] R. Jung, H.-H. Dang, J. Kang, and D. Dreyer, "Stacked borrows: an aliasing model for rust," *Proceedings of the ACM on Programming Languages*, vol. 4, no. POPL, pp. 1–32, 2019.

[68] Rust Foundation, "Asynchronous programming in rust." https://rust-lang.github.io/async-book/02_execution/01_chapter.html, 2024. Accessed: 2024-02-28.

[69] Cognitive Engineering Lab, "The rust programming language - rust avoids simultaneous aliasing and mutation." `https://rust-book.cs.brown.edu/ch04-02-references-and-borrowing.html#rust-avoids-simultaneous-aliasing-and-mutation`, 2024. Accessed: 2024-02-18.

[70] R. Likert, "A technique for the measurement of attitudes.," *Archives of psychology*, 1932.

[71] A. Rossberg, "Webassembly specification," *WebAssembly Community Group*, vol. 2, 2021.

[72] C. Schott, "Boris - an ownership and borrowing visualizer." `https://users.rust-lang.org/t/boris-an-ownership-and-borrowing-visualizer/105877`, 2024. Accessed: 2024-02-21.

[73] C. Schott, "Boris - an ownership and borrowing visualizer." `https://www.reddit.com/r/rust/comments/1abrcr2/boris_an_ownership_and_borrowing_visualizer/`, 2024. Accessed: 2024-02-21.

[74] C. Schott, "Boris - a visualizer for rust's ownership and borrowing mechanics." `https://www.reddit.com/r/learnprogramming/comments/1bg9y65/boris_a_visualizer_for_rusts_ownership_and/`, 2024. Accessed: 2024-03-27.

A Appendix

Section A.1 contains the contents of the survey adapted from the survey website.

A.1 Survey - Visualizing Ownership and Borrowing in Rust Programs

About you

Overall Experience level Programming

- Beginner: fundamental knowledge
- Student: actively learning
- Professional: working in the industry for >1 year
- Expert: working in the industry for >5 years

Select the types of languages you are familiar with/ use regularly

Languages you are comfortable using for personal or professional projects, even if you would not consider yourself being an expert.

- Dynamically Typed Languages (e.g., Python, JavaScript)
- Garbage Collected Languages (e.g., Java, C#)
- Low Level Languages (e.g., C, C++)
- Functional Languages (e.g., Haskell, Scala, OCaml)
- Other (comment below)

How familiar are you with the Rust programming language?

- Never have heard of it before
- Heard of it, but never used it
- In the process of learning it
- Used it for small personal toy projects
- Used it for more 'serious' projects (actually released something)
- Expert (>2 years of professional/ in production use)
- Other (comment below)

Rust

is a modern programming language recognized for prioritizing safety, reliability, and performance. It features a unique ownership system for memory management that prevents common programming errors, making it well-suited for building robust and efficient software.

Here is a short overview over Rust's memory management model utilizing interactive visualizations.

Ownership

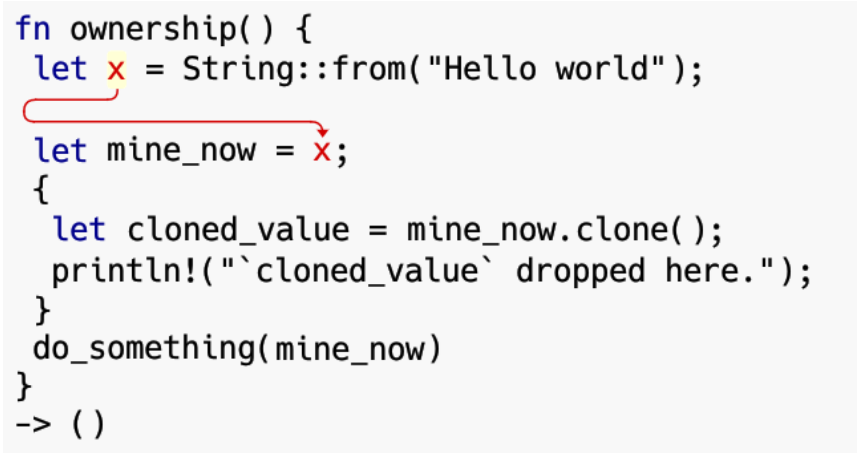
In Rust, you can bind a value to an owner using the statement:

```
let owner = value;
```

Rules:

- Each value in Rust has an owner.
- There can only be one owner at a time.
- When the owner goes out of scope, the value will be dropped (and its memory freed).

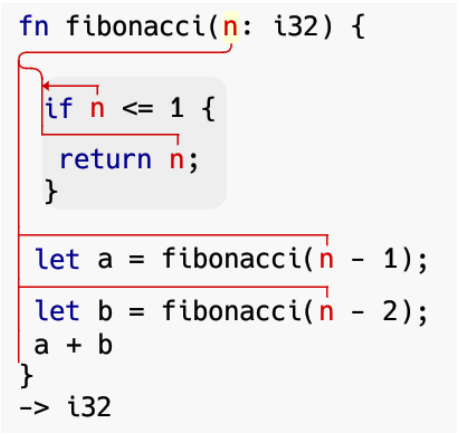
Click on variables to visualize an ownership/ borrowing graph.



- the first line assigns ownership over the string "Hello world" to `x`
- in the second line, the ownership over the string is moved to `mine_now`
- accessing `x` after that would be a compile error

Primitive Types

One exception to these ownership rules are primitive values (e.g., numbers). When a primitive value is used, its value is implicitly copied, instead of moving its ownership.

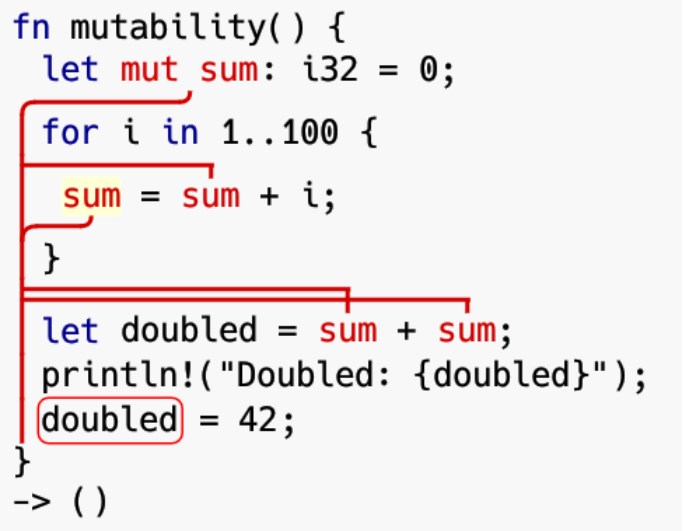


Note how `n` is used in multiple places, without transferring ownership of its value.

Mutability

By default, all variables in Rust are readonly. To mutate/write to an existing value, its owner must be marked mutable (`mut`) explicitly.

Thicker lines indicate mutable values.



Note that the last line, would cause a compile error, as doubled is not mutable.

Borrowing

Instead of passing ownership to a value, we can also borrow the value from its owner with the `&`-operator.

When the owner is marked as mutable, we can also create a mutable reference (`&mut`) to the value (which allows writing to it).

Rules:

- At any given time, you can have **either one mutable** reference **or any number of immutable** references to a value.
- References must always be valid, so the value must not be moved while it is referred to.

```
fn borrowing( ) {
 println!("Multiple borrows to an immutable string.");
 {
   let owned_string = String::from("Hello");
   let borrow = &owned_string;
   borrow_string(&owned_string);
   borrow_string(borrow);
 }
 let mut mutable_string = String::from("World");
 if mutable_string.is_empty() {
  mutable_string.push_str("Hello there.");
 }
 else {
  println!("The string is not empty.");
  modify_string(&mut mutable_string);
 }
 println!("'{}'", mutable_string);
}
-> ()
```

These are the basic principles of Rust's memory management (more information can be found here [27]).

Please rate the following statements about the visualizations

1. strongly disagree
2. disagree
3. neutral
4. agree
5. strongly agree

- **They are intuitive to use and grasp** Would you detect errors faster compared to console-based borrow checker errors?
- **They helped you to understand Ownership and Borrowing** or deepened your (intuitive) understanding if you were already familiar with these concepts.
- **They could help to explain these concepts to Beginners** Making Rust easier and faster to learn for new programmers, and ones switching from other languages.
- **You would use such Visualizations during development** Assuming they were directly integrated into your IDE.

Questions, Comments, Suggestions?

Julius-Maximilians-
UNIVERSITÄT
WÜRZBURG

Versicherung zur selbständigen Leistungserbringung

Titel der Masterarbeit:

Visualizing Ownership and Borrowing in Rust Programs

Thema bereitgestellt von (Titel, Vorname, Nachname, Lehrstuhl):

Prof. Dr.-Ing., Samuel, Kounev, Chair of Software Engineering, Department of Computer Science

Eingereicht durch (Vorname, Nachname, Matrikel):

Christian, Schott, 2334089

Ich versichere, dass ich die vorstehende schriftliche Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Die benutzte Literatur sowie sonstige Hilfsquellen sind vollständig angegeben. Wörtlich oder dem Sinne nach dem Schrifttum oder dem Internet entnommene Stellen sind unter Angabe der Quelle kenntlich gemacht.

Weitere Personen waren an der geistigen Leistung der vorliegenden Arbeit nicht beteiligt. Insbesondere habe ich nicht die Hilfe eines Ghostwriters oder einer Ghostwriting-Agentur in Anspruch genommen. Dritte haben von mir weder unmittelbar noch mittelbar Geld oder geldwerte Leistungen für Arbeiten erhalten, die im Zusammenhang mit dem Inhalt der vorgelegten Arbeit stehen.

☒ Mit dem Prüfungsleiter bzw. der Prüfungsleiterin wurde abgestimmt, dass für die Erstellung der vorgelegten schriftlichen Arbeit Chatbots (insbesondere ChatGPT) bzw. allgemein solche Programme, die anstelle meiner Person die Aufgabenstellung der Prüfung bzw. Teile derselben bearbeiten könnten, entsprechend den Vorgaben der Prüfungsleiterin bzw. des Prüfungsleiters eingesetzt wurden. Die mittels Chatbots erstellten Passagen sind als solche gekennzeichnet.

Der Durchführung einer elektronischen Plagiatsprüfung stimme ich hiermit zu. Die eingereichte elektronische Fassung der Arbeit ist vollständig. Mir ist bewusst, dass nachträgliche Ergänzungen ausgeschlossen sind.

Die Arbeit wurde bisher keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht. Ich bin mir bewusst, dass eine unwahre Erklärung zur Versicherung der selbstständigen Leistungserbringung rechtliche Folgen haben kann.

Würzburg, 07.05.2024 C Schott

Ort, Datum, Unterschrift