The Usability of Ownership

WILL CRICHTON, Stanford University

Ownership is the concept of tracking aliases and mutations to data, useful for both memory safety and system design. The Rust programming language implements ownership via the borrow checker, a static analyzer that extends the core type system. The borrow checker is a notorious learning barrier for new Rust users. In this paper, I focus on the gap between understanding ownership in theory versus its implementation in the borrow checker. As a sound and incomplete analysis, compiler errors may arise from either ownership-unsound behavior or limitations of the analyzer. Understanding this distinction is essential for fixing ownership errors. But how are users actually supposed to make the correct inference? Drawing on my experience with using and teaching Rust, I explore the many challenges in interpreting and responding to ownership errors. I also suggest educational and automated interventions that could improve the usability of ownership.

ACM Reference Format:
Will Crichton. 2020. The Usability of Ownership. In *Proceedings of Human Aspects of Types and Reasoning Assistants (HATRA '20).* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Ownership is nominally quite simple to explain. It has three basic rules:

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

Statically checking these three rules ensures that deallocated memory can never be accessed at runtime. It even rules out unsynchronized concurrent mutation (data races). However, like most interesting program properties, checking whether an arbitrary well-typed Rust program satisfies ownership is undecidable. Instead, Rust uses a sound and incomplete algorithm to verify ownership in a subset of well-typed Rust programs, inspired by prior work in region-based memory management [3, 10].

The purpose of this paper is to explore the human factors of ownership-oriented programming in the face of incompleteness. How does a programmer learn to deal with the borrow checker? How can they build an accurate mental model of what the type system will accept and reject? And how can they transfer knowledge from other languages into Rust?

2 A LITANY OF ERRORS

To get a sense of the problem, let's say a programmer is learning how to use the vector data type. We start with a constructor:

```rust
let mut v = vec![1, 2, 3];
```

Under Rust 1.47 (stable as of November 2020), if we try to compile this code:

HATRA '20, November 18-19, 2020, Chicago, IL
© 2020 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM...$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

```
let x = &v[0];        // Get an immutable pointer to first vector element
v.push(1);            // Attempt to mutate vector
assert!(*x == 1);     // Read from immutable pointer after mutation
```

Then the Rust compiler will raise the following error:

```
error[E0502]: cannot borrow `v` as mutable because it is also borrowed as immutable
1 |   let x = &v[0];
  |            - immutable borrow occurs here
2 |   v.push(1);
  |   ^^^^^^^^^ mutable borrow occurs here
3 |   assert!(*x == 1);
  |           -- immutable borrow later used here
```

This catch is an objective win — we avoided an iterator-invalidation scenario where the push reallocates the vector, leaving x pointing to freed memory. Here's a murkier example:

```
let mut x = &mut v[0]; // Get a mutable pointer to first element
let mut y = &mut v[0]; // Get another mutable pointer to first element
*y += *x;              // Attempt to dereference x and y
```

The compiler will against raise an error:

```
error[E0499]: cannot borrow `v` as mutable more than once at a time
1 |   let mut x = &mut v[0];
  |                    - first mutable borrow occurs here
2 |   let mut y = &mut v[0];
  |                    ^ second mutable borrow occurs here
3 |   *y += *x;
  |         -- first borrow later used here
```

This program is *technically* memory-safe, but that's contingent on our use of integer data types. If the vector contained vector elements (`Vec<Vec<i32>>`) then we would have two mutable pointers to the same vector, which could lead to an iterator invalidation scenario like above. It makes sense that this behavior should be generally disallowed. Now let's change the indices to refer to different elements.

```
let mut x = &mut v[0];
let mut y = &mut v[1]; // Now a mutable pointer to a different element
*y += *x;
```

The compiler will raise the exact same error as before. However, this program is provably safe regardless of data type. Because a vector owns its elements, `v[0]` and `v[1]` are guaranteed not to alias. Yet, Rust will reject the program due to incompleteness: it conservatively assumes any indexing operation could refer to any element. The only way to implement this pattern in Rust is to use raw pointers in an `unsafe` block, or to use the specific stdlib function `Vec::split_at_mut` (assuming the programmer is aware of it).

In some cases, statements that look virtually identical at the type-level can be treated differently. For example:

```
    v.insert(0, v[0]);   // Read v[0] and insert it as the 0-th element
    v.get_mut(v[0]);     // Read v[0] and get a mutable pointer to that index
```

Both of these statements use a method that implicitly take a mutable reference to v (i.e. `&mut self`). However, reading `v[0]` constitutes taking an immutable reference to v. Today on Rust 1.45, only the second line fails to compile with the error:

```
error[E0502]: cannot borrow `v` as immutable because it is also borrowed as mutable
1 |   v.get_mut(v[0]);
  |   - ------- ^ immutable borrow occurs here
  |   | |
  |   | mutable borrow later used by call
  |   mutable borrow occurs here
```

About two years ago on Rust 1.30, *both* of these lines would fail to compile. This disparity is due to Rust's new "non-lexical lifetimes" (NLL) algorithm within the borrow checker. NLL increased the overall completeness of the type system, but not always in easily predictable ways. Here, the difference is that `get_mut` is actually defined on slices (which vectors can implicitly convert into) while `insert` is defined only on vectors. For whatever reason, that distinction is meaningful enough to cause one line to error and the other to not.

Here's another example of two semantically equivalent statements:

```rust
// Two examples of: get first element, pushing 1 if not exists
match v.get(0) { Some(x) => x, None => { v.push(1); &v[0] } };
v.get(0).unwrap_or_else(|| { v.push(1); &v[0] });
```

Again, both lines fail to compile on Rust 1.30 (without NLL), and only the second line fails to compile on Rust 1.44 (with NLL). The reason being that the usage of a closure in the second line restricts the borrow checker to assuming a longer lifetime for `&v[0]` than without a closure in the first line.

3 THE CURSE OF INCOMPLETENESS

The examples show that building a mental model of how ownership *actually* works in Rust is hard. In my experience, these scenarios are all realistic problems that a Rust programmer may encounter. They explain why the borrow checker is consistently a top reason why developers struggle with Rust or give up on it entirely [9]. Research on the usability of ownership has similarly found that programmers "were not prepared to use a type system to address an [ownership] bug that they thought of in a dynamic way" [2].

A core usability problem is the gap between the simplicity of the goal (three ownership & borrowing rules) and the complexity of the implementation (borrow checker). I can teach the three rules in a single lecture to a room of undergrads. But the vagaries of the borrow checker still trip me up every time I use Rust! When faced with a borrow checker error, I ask the following questions in order:

(1) Does my code actually violate the ownership rules?
(2) Can I restructure my code to satisfy the borrow checker?
(3) Is there a function that can bypass the borrow checker?
(4) Do I need to write that unsafe code myself?

I'd like to talk about each question in greater detail: what makes this question hard to answer, and what resources and pedagogies can help?

3.1 Does my code actually violate the ownership rules?

The first challenge is to determine whether an error is caused by unsoundness in the program or incompleteness in the borrow checker. Before all else, a programmer must be aware that this distinction exists. Learning resources about the borrow checker rarely emphasize this fact — the official Rust Book [4], for example, just teaches the high-level ownership/borrowing rules.

Consider a Rust novice with C++ experience trying to transfer patterns from one language to another. To read a string from a vector, they can write a valid C++ program:

```
vector<string> v = {"a", "b", "c"};
string s = v[0];
cout << s << endl;
```

Then, they try a simple rewriting into Rust:

```
// .into() syntax converts &'static str pointer to owned String
let v: Vec<String> = vec!["a".into(), "b".into(), "c".into()];
let s = v[0];
println!("{}", s);
```

This code fails to compile with the error:

```
error[E0507]: cannot move out of index of `std::vec::Vec<std::string::String>`
2 |   let s = v[0];
  |           ^^^^
  |           move occurs because value has type `std::string::String`,
  |               which does not implement the `Copy` trait
```

The subtle difference between the two programs is that line 2 of the C++ program implicitly invokes the string's copy constructor, causing a heap allocation. Rust only implicitly copies types that implement the `Copy` trait (e.g. integers, pointers) that are bit-for-bit copyable without invoking the heap allocator. This distinction is fundamental to ownership — a Rust programmer must unlearn implicit copying patterns acceptable in other languages. Here, they either need to take a reference (`&v[0]`) or deep-copy the data (`v[0].clone()`).

However, the programmer later tries to mutably index disjoint parts of the array, e.g. `&mut v[0]; &mut v[1]` like above. This pattern is also acceptable in C++, but nominally disallowed in Rust. The Rust novice may incorrectly assume that this limitation is fundamental to sound ownership, not understanding that the borrow checker's rejection is purely a matter of incompleteness. Resources for teaching Rust should consider and explicit focus on the sound vs. incomplete distinction, and giving examples of when borrow checker errors fall into one category or another.

3.2 Can I restructure my code to satisfy the borrow checker?

If the programmer has decided their code is actually sound, the next step is to move the program out of the incompleteness zone by small semantics-preserving transformations. In this step lies a dark, perverse bag of tricks that all Rust programmers must learn. For example, consider this problematic code from earlier:

```
v.get_mut(v[0]);
```

We can appease the borrow checker by simply binding the argument to a variable:

```
let x = v[0];
v.get_mut(x);
```

As a more complex example, let's say we want to build an index over a document that maps each token to a list of occurrences. Before building the index, we apply preprocessing to filter unwanted tokens.

```
struct Index {
  tokens: Vec<String>,
  index: HashMap<String, Vec<usize>>
}

impl Index {
  fn nonempty_tokens(&self) -> Vec<&String> {
    // Get a vector of *pointers* to strings in self.tokens
    self.tokens.iter().filter(|s| s != &"").collect()
  }

   fn build_index(&mut self) {
    // Mutate self.index while iterating over vector of string pointers
    for (idx, token) in self.nonempty_tokens().into_iter().enumerate() {
      let entry = self.index.entry(token.clone()).or_insert(vec![]);
      entry.push(idx);
    }
  }
}
```

This program fails to compile with the error:

```
error: cannot borrow `self.index` as mutable because it is also borrowed as immutable
14 |     for (idx, token) in self.nonempty_tokens().iter().enumerate() {
   |                         -----------------------------------------
   |                         immutable borrow occurs here
15 |       let entry = self.index.entry((*token).clone()).or_insert(vec![]);
   |                   ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ mutable borrow occurs here
```

The issue is that nonempty_tokens returns an immutable reference to values within the tokens vector. However, the borrow checker reasons conservatively across function boundaries, so an immutable borrow to any field is considered an immutable borrow on the entire struct. Hence, when attempting to mutably update self.index in build_index, the compiler complains of a mutable/immutable conflict.

This kind of pattern presents a challenging design problem. One workaround is to inline the nonempty_tokens function. For example, if we write:

```
let nonempty_tokens = self.tokens.iter().filter(|s| s != &"").collect();
for (idx, token) in nonempty_tokens.into_iter().enumerate() { .. }
```

This program will pass the borrow checker, because it's able to reason locally about the independence of the tokens and index borrows. But, this approach reduces the modularity of our program. Another alternative is to push the nonempty_tokens method down onto the self.tokens field.

```
struct Tokens(Vec<String>);
impl Tokens {
  fn nonempty(&self) -> Vec<&String> { self.0.iter().filter(..).collect() }
}

struct Index { tokens: Tokens, .. }
impl Index {
  fn build_index(&mut self) {
    for (idx, token) in self.tokens.nonempty().into_iter().enumerate() { .. }
  }
}
```

The high-level point is that these tricks are not obvious — they are a form of tacit knowledge that ought to be explicitly taught. Moreover, to understand why the tricks work and when to apply them, a programmer must understand the internals of the borrow checker. To answer "is my code sound?", a programmer needs to only identify a disparity between the high-level rules and a black-box incomplete analyzer. But to actually satisfy the borrow checker (without unsafe-based back-doors), a black-box is no longer sufficient.

Consequently, we need to teach Rust programmers about the actual implementation of the borrow checker, not just the rules. Herein lies an interesting research question: is a more complex analysis (i.e. more complete) on balance easier or harder for programmers to use? In theory, a more complex/complete analysis means fewer situations to wrestle with the analyzer, but a greater challenge when those situations arise. Interestingly, the Rust developers claimed the introduction of non-lexical lifetimes was easier to understand, despite the increase in complexity. Quoting the feature proposal [6]:

> Part of the reason that Rust currently uses lexical scopes to determine lifetimes is that it was thought that they would be simpler for users to reason about. Time and experience have not borne this hypothesis out: for many users, the fact that borrows are "artificially" extended to the end of the block is more surprising than not. Furthermore, most users have a pretty intuitive understanding of control flow. [...]
>
> The primary [drawback] is that the rules for the system become more complex. However, this permits us to accept a larger number of programs, and so we expect that using Rust will feel simpler. Moreover, experience has shown that — for many users — the current scheme of tying reference lifetimes to lexical scoping is confusing and surprising.

3.3 Is there a function that can bypass the borrow checker?

Prior work on barriers to adoption for Rust has analyzed dozens of blog posts and forum threads documenting programmers' experiences with learning the language [11]. One common interaction had the form:

- **Newbie**: I can't figure out how to do X! The borrow checker rejects my code, and I can't find any answer about X online.
- **Expert**: Oh, that's because you can't technically do X without `unsafe`. Did you know about library function Y? It does exactly what you need.
- **Newbie**: Wow! That's perfect. If only I had known...

For example, X = mutably referencing disjoint parts of an array, and Y = `Vec::split_at_mut`. Or X = temporarily taking ownership of a value of type `&mut T`, and Y = `take_mut::take`.

The core issue here is discoverability. In many of the interactions we observed, novices weren't aware of the sound/incomplete distinction, so they didn't think to go looking for a helper function. Even if they did go searching, finding the right function poses a big challenge. For example, if you wanted to find `Vec::split_at_mut` but didn't know the method name, the Rust documentation for `Vec` lists 182 methods. Good luck!

I've had echoes of this experience when learning to write proofs in Lean. If I'm writing a proof about natural numbers and my tactics fail, I'll start hunting for a relevant theorem. It's an excruciating experience to carefully scan the hundreds of theorems in the standard library, until I find just the right `nat.pow_lt_pow_of_lt_right` or whatever.

One basic strategy is to create textual resources and automated lints that help identify when users encounter these issues, and point to possible fixes. For example, steban Küber is. the unsung hero of Rust who works on the compiler's error messages. After publication of the Rust barriers to adoption paper [11], he implemented a help message for rustc that detects the specific pattern of disjoint mutable indexing of an array, and points out `Vec::split_at_mut` as a possible solution.

Here's another research challenge: can we save Esteban the time and automatically generate these lints?

3.4 Do I need to write that unsafe code myself?

When all else fails, it's time to program like it's 1972: with `unsafe` and raw pointers. While uncommon in pure Rust code, these situations do arise. Consider the `Index` example from above. The hash map used owned strings as keys, which requires an a new heap allocation per key. A more efficient strategy would use to use string pointers as keys, something like:

```
// 'a is a lifetime variable, NOT a type variable
// 'a is a lower bound on the lifetime of the index string pointers
struct Index<'a> {
  tokens: Vec<String>,
  index: HashMap<&'a str, Vec<usize>>
}
```

This example is a *self-referential struct*, i.e. one field contains pointers to another field. In general, this pattern is unsound. If the `tokens` field is moved or dropped, the `&'a str` pointers become invalid. The Rust community has developed user-level abstractions that help with these patterns, e.g. the `Pin` API that ensures an object will never be moved. However, when designed at the user-level, the compiler isn't aware of these properties, and so `Pin` must be used in conjunction with raw pointers.

Beyond the obvious safety concerns of dealing with raw pointers (see [7] for gory details), the usability challenge is that unsafe code is an entirely new sub-language to learn. Unsafe programming includes new data structures (`Pin`, `UnsafeCell`, `NonNull`), pointer manipulation APIs (`std::ptr`, `std::mem`), and general best practices to avoid self-inflicted foot injuries. Conversely, some of the implicit affordances of Rust's type system fall away — pointers are no longer automatically dereferenced, for example.

Another concern is whether novices will put in the effort to identify situations where `unsafe` is necessary. Several people have asked me over the years, "if Rust lets people write `unsafe`, won't everyone just get lazy and end up being unsafe?" This concern is similar to gradually-typed languages like Typescript, where it's just *so easy* to banish a type error with a little "`any`" annotation. For Rust, I'm optimistic that unsafe is sufficiently hard to use and discouraged by the community, such that a novice won't reach for it unless they absolutely need to. Nonetheless, when the time comes for unsafe, Rust users face an uphill battle in learning an unfamiliar language.

4 TO OWNERSHIP AND BEYOND

In summary, a programmer trying to understand ownership in Rust must learn the following:

(1) The ownership rules, separated from how they're enforced by the borrow checker.
(2) The distinction between soundness and completeness.
(3) How the borrow checker actually works, and in what situations it's likely to fail.
(4) That functions may exist to make the unsafe into safe, and how to find them.
(5) How to identify when unsafe is absolutely necessary.

I think just studying these issues, specifically in the context of Rust and its borrow checker, would be a wonderful part of the research agenda for the HATRA community. After all, Rust is the single most widely-used functional programming language *ever*. More than Haskell, Scala, OCaml and Elm combined! The Rust community has hundreds of thousands of people to be both studied and supported in our research.

But beyond Rust and beyond ownership, I think the challenges described here can provide a useful framework for investigating the usability of other type systems. Whether it's dependent types, effect systems, or anything else, we will always have (hopefully) sound and (definitely) incomplete static analyses. Programmers using these systems will need to distinguish between the kind of type errors they receive, and learn to use or avoid type system back-doors.

Solutions to this problem will necessarily come from many domains. Here's a few more research problems to think about:

- **Notional machines:** Researchers in CS education have developed a theory of notional machines, or how students build a mental model of a language's operational semantics [8]. What would a notional machine for the borrow checker look like? Perhaps we could present formulations of type systems that look more like runtime semantics [5].
- **Visualizations:** type systems like Rust's infer a significant amount of static information, like types and lifetimes. Tools for visualizing this information can help programmers build a mental model of how the type checker works, and reason about why individual errors occur. Displaying this information in a succinct, non-intrusive, yet informative way is still an open problem.
- **Error messages:** besides visualization, the other kind of human-compiler interface is an error message. Errors are responsible for helping programmers localize and interpret syntax/type bugs. Prior work in improving errors has focused on solving the programmer's immediate problem [1]. But can error messages also be used pedagogically, to teach the programmer how to avoid a class of type errors in the future?

REFERENCES

[1] Brett A Becker, Paul Denny, Raymond Pettit, Durell Bouchard, Dennis J Bouvier, Brian Harrington, Amir Kamil, Amey Karkare, Chris McDonald, Peter-Michael Osera, et al. 2019. Compiler Error Messages Considered Unhelpful: The Landscape of Text-Based Programming Error Message Research. In *Proceedings of the Working Group Reports on Innovation and Technology in Computer Science Education*. 177–210.

[2] Michael Coblenz, Gauri Kambhatla, Paulette Koronkevich, Jenna L Wise, Celeste Barnaby, Joshua Sunshine, Jonathan Aldrich, and Brad A Myers. 2019. Usability Methods for Designing Programming Languages for Software Engineers. *arXiv* (2019), arXiv–1912.

[3] Trevor Jim, J Gregory Morrisett, Dan Grossman, Michael W Hicks, James Cheney, and Yanling Wang. 2002. Cyclone: A Safe Dialect of C.. In *USENIX Annual Technical Conference, General Track*. 275–288.

[4] Steve Klabnik and Carol Nichols. 2020. The Rust Programming Language. https://doc.rust-lang.org/book/title-page.html

[5] George Kuan, David MacQueen, and Robert Bruce Findler. 2007. A rewriting semantics for type inference. In *European Symposium on Programming*. Springer, 426–440.

[6] Niko Matsakis. 2017. RFC 2094: non-lexical lifetimes. https://github.com/rust-lang/rfcs/blob/master/text/2094-nll.md

[7] Boqin Qin, Yilun Chen, Zeming Yu, Linhai Song, and Yiying Zhang. 2020. Understanding memory and thread safety practices and issues in real-world Rust programs. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 763–779.

[8] Juha Sorva. 2007. Notional machines and introductory programming education. *Trans. Comput. Educ* 13, 2 (2007), 1–31.

[9] Rust Survey Team. 2018. Rust Survey 2018 Results. https://blog.rust-lang.org/2018/11/27/Rust-survey-2018.html

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

[11] Anna Zeng and Will Crichton. 2018. Identifying Barriers to Adoption for Rust through Online Discourse. In *9th Workshop on Evaluation and Usability of Programming Languages and Tools (PLATEAU 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.