Ownership types in theory and practice (in Rust)

Fritz Rehde
Technical University of Munich
Germany
fritz.rehde@tum.de

Abstract

Object aliasing is the concept of accessing the same memory through different symbolic names in object-oriented programming languages. Many programming bugs are created through unintentional aliases, which are hard to detect and can lead to unexpected side effects. Ownership types are one solution that attempts to prevent many alias-related bugs. The premise of ownership types is that not only the fields of an object are protected from external access, but also all objects stored in those fields. This is done by allowing objects to take ownership of other objects.

This paper depicts some of the different kinds of ownership types, explains how the modern programming language Rust uses ownership types and evaluates Rust's implementation by comparing its use of the ownership concept to that of another popular systems programming language, namely C++.

Keywords Ownership, Type, Safety, Rust

1 Introduction

One of the main goals of ownership types is to improve memory safety. Understanding memory-related bugs will, therefore, help in understanding why ownership types are useful.

1.1 Memory Safety

Memory safety refers to the state of a program where memory pointers or references always refer to valid memory.

The tech giants Google [15] and Microsoft [7] have both revealed that around 70 percent of their vulnerabilities were the result of memory safety issues. The importance of memory safety is further underlined by the fact that the National Security Agency of the United States of America has published a Cybersecurity Information Sheet [13] on the topic, the contents of which are described in the following section.

There are a variety of potential occurrences of memory management issues. Examples include *buffer overflows*, where data is accessed outside the bounds of an array, *memory leaks*, where the programmer forgets to free memory that has been allocated, causing the program to, eventually, run out of memory, *Use-After-Free*, where memory is accessed after it has been freed, *Double-Free*, where memory is freed again after it had already been freed before, and the use of uninitialized memory. These issues can not only enable potential malicious exploits, but they can also result in incorrect program results, the decrease of a program's performance or seemingly random program crashes.

Commonly used programming languages, such as C and C++, provide a lot of freedom and flexibility in memory management, but this comes at the cost of heavily relying on the programmer to perform the needed checks on memory references. Furthermore, the programmer must perform rigorous testing to ensure that the software handles surprising conditions. Even though software analysis tools can detect many instances of memory management issues, the NSA recommends the use of inherently memory-safe languages.

Example Ralf Jung [11] provides the following common example of how ownership types can prevent a memory safety issue.

First, we will explore how a memory safety problem is created in the following C++ code snippet that does not use the ownership concept.

```cpp
std::vector<int> v { 10, 11 };
int *vptr = &v[1]; // points into v
v.push_back(12);
std::cout << *vptr; // bug (use-after-free)
```

Listing 1. C++ use-after-free [11]

We initialize a vector *v* that contains two integers stored in a buffer in memory. Next, we create a pointer *vptr* that points into this buffer, specifically to the place where the second element (with current value 11) is stored. Now, both *v* and *vptr* point to (overlapping parts of) the same buffer. We say that the two pointers are aliasing. Then, we push a new element to the end of *v*. If the vector's capacity were large enough, the new element would be appended to the buffer. However, we will assume there is no more space for an additional element, so a new buffer is allocated, and all the existing elements are moved over. This case is interesting because *vptr* still points to the old buffer! In other words, adding a new element to *v* has turned *vptr* into a dangling pointer. Therefore, trying to access the value, which the dangling pointer *vptr* points to, will cause a use-after-free bug. More generally, we can observe that an action through a pointer (*v*) will also affect all of its aliases (*vptr*), even though these aliases might not expect a change. The main problem here is that such memory issues cannot be detected at compile time.

Now, we will demonstrate how ownership types in Rust allow the compiler to detect such a use-after-free bug. Consider the following Rust translation of our C++ example:

```
1 let mut v = vec![10, 11];
2 let vptr = &mut v[1]; // points into v
3 v.push(12);
4 println!("{}", *vptr); // compiler error
```

Listing 2. Rust use-after-free [11]

Syntactically, the C++ and Rust versions are very similar. Notably, the *push* method in line 3 has the following signature: `pub fn push(&mut self, value: T)` [4]. While the C++ version will compile successfully, causing a run-time security vulnerability, the Rust compiler shows an error: "Cannot borrow *v* as mutable more than once at a time." For now, it suffices to say that the Rust compiler disallows both lines 2 and 3 to create mutable references to *v* if one of the mutable references is used later (like *vptr* is in line 4). In 3.5 and 3.5, we will explain in more detail how the Rust borrow-checker is able to detect this problem using Rust's ownership rules.

1.2 State of the art

The concept of ownership types provides one of the solutions aiming to eliminate many of the described memory-related bugs. Potential solutions to the aliasing problem include banning aliases altogether, clearly advertising aliases or managing and controlling their effects [8]. Many different flavors of ownership types have been explored. In this paper, state-of-the-art implementations of the ownership types concept, such as owners-as-dominators and owners-as-modifiers, will be explored further. These approaches differ in what kind of, if any, aliases they allow.

2 Ownership types in theory

In object-oriented programs, an object can reference any other object and read and modify its fields through direct field accesses or method calls. Such programs with arbitrary object structures are difficult to understand, maintain, and reason about. [10]

Ownership types can limit which objects can be referenced, and can specify whether the referenced objects may be mutated or just read from.

If not otherwise indicated, all of the information gathered in this chapter stems from the Ownership Types Survey [8].

2.1 Domain specific terms

In the following, some terms from the Ownership Types survey [8], which are commonly used in this chapter, are defined.

The core concept of ownership types is that each object is *owned* by (at most) one other object, called its *owner*. Objects are organized into different *ownership contexts*. An ownership context is the set of all objects with the same owner. Objects that are in the same ownership context, meaning they share the same owner, are called *siblings* or *peers*. More informally, an object's ownership context can also be visualized as a box into which all of its owned objects are placed, to store and protect these objects. Each object is considered to be *inside* the object whose ownership context box it is placed in. There also exists a more formal definition of the *inside* relation: Given the two objects *a* and *b*, *a* is *inside* *b* if *a* is the same object as *b* or *a* is *owned* by *b*, transitively. The *outside* relation is the converse of the *inside* relation. This hierarchy induces a nesting relationship between objects. [8]

2.2 Owners-as-dominators

One flavor of ownership types is the *owners-as-dominators* model [8], which is an implementation of the core encapsulation mechanism from the original Flexible Alias Protection concept [9].

Zhao and Boyland [17] specify that, with owners-as-dominators, each regular object is owned by exactly one object and can be the owner of zero or more objects.

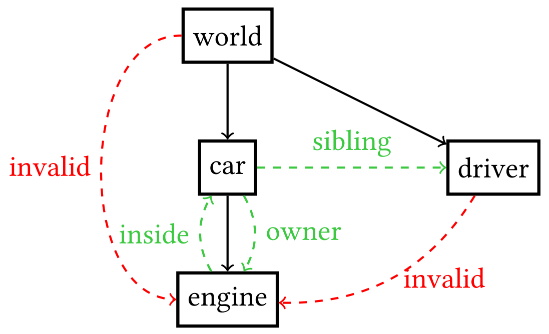

Figure 1. Owners-as-dominators [9]: Solid lines indicate "owns", dotted lines indicate "references".

Objects can also be owned by a special global object called *world*, which cannot be owned anymore. This *world* object is the root of the acyclic ownership hierarchy tree that represents the ownership relation. In this tree structure, each object is inside its owner. This model is named *owners-as-dominators* because any object acting as an owner is considered to be a *dominator* for all of its owned objects.

The ownership survey [8] defines that at least one of the following rules must apply for object *a* to validly reference *b*.

1. *a* is the owner of *b*,
2. *a* and *b* are siblings, or
3. *b* is outside of *a*.

Figure 1 was adapted from an example by Clarke et al. [9] and demonstrates valid (green) and invalid (red) references in an ownership hierarchy. Each of the three reference-validity rules that were defined previously will be illustrated using this example figure.

Owner The object *car* owns the object *engine*, meaning that *car* can validly reference *engine* according to the first rule.

An owner being able to access its owned objects is the most trivial and intuitive rule.

Sibling The objects *car* and *driver* are siblings because they are both in the ownership context of the object *world*. This means that they can validly reference each other according to the second rule.

The lack of further rules implies that siblings are the only objects that an object can validly reference besides the objects it owns and the objects it is inside of. The sibling relationship would, therefore, be used for objects that exist independently but should still be able to interact with each other.

Outside/inside The object *engine* is inside the object *car* because *engine* is owned by *car*. Conversely, *car* is outside *engine*. Therefore, according to the third rule, *engine* can validly reference *car*.

The original Flexible Alias Protection model, on which these ownership types are based, imposed the same restriction. "The only objects that can access [an object] are the object that owns it, and other objects inside that [object]" [9]. The implication is that this model only protects objects from external access. However, internally, an object can still access its owner (and its owner's owner etc.).

Dietl and Müller [10] point out that there exist several slightly different specifications of the owners-as-dominators model. They claim that Clarke's owners-as-dominators property [8] is weaker than others "by allowing instances of inner classes to access the representation of the instance of the outer class they are associated with" [10], which we have already identified in the third rule. They state that, thereby, this owners-as-dominators model can handle iterators, but not more general forms of sharing [10].

Further implications Primarily, these three rules imply that any external reference to an object is only possible through its owner. This is exemplified in Figure 1 where the object *world* is trying to access the object *engine*, which is not allowed because *world* is neither a direct owner of, a sibling of or inside of *engine*. Instead, *world* could only reference *engine* through the *engine*'s owner *car*.

According to Dietl and Müller [10], the owners-as-dominators model restricts *where* references are allowed to point to. Any object that can be validly referenced may be mutated through that reference.

Its strictness can be seen as both an advantage and a disadvantage of the owners-as-dominators model. The advantage is that the model provides a simple, clear and strong guarantee that allows for reasoning about various properties of the code. However, it also makes programming more difficult by not allowing common idioms that involve aliasing. [8]

2.3 Owners-as-modifiers

Another kind is the *owners-as-modifiers* model [8], which is a weaker form of owners-as-dominators that additionally allows read-only references. A read-only reference can only be used to read fields and call *pure methods*. Pure methods are defined as methods that do not modify the fields of any existing objects. [8]

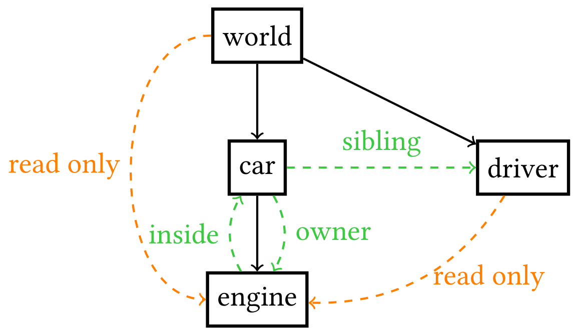

Figure 2. Owners-as-modifiers [8]: Solid lines indicate "owns", dotted lines indicate "references".

For *a* to validly reference *b* through reference *r*, at least one of the following rules [8] must apply.

1. *a* is the owner of *b*,
2. *a* and *b* are siblings, or
3. *b* is outside of *a*, or
4. *r* is a read-only reference and only pure methods can be called on it.

We can see that the owners-as-modifiers rules extend the rules specified for owners-as-dominators. The only additional rule is that a reference can also be valid if it is a read-only reference and only pure methods are being called on it.

Implications Dietl and Müller [10] suggest that the owners-as-modifiers approach allows references to point to objects in arbitrary contexts, but restricts *how* references can be used. In particular, any object can be referenced by any given object. However, only an object that is the owner of, a sibling of or inside of another given object may receive a reference to that object that allows mutating the object. Figure 2 demonstrates that all other objects, such as *world* or *driver*, which may not be allowed to obtain a mutable reference to an object for the reasons above, can still receive read access to said object.

The advantage of this model is that it expands the programming possibilities of the owners-as-dominators principle and fulfills the requirements needed for the verification of the functional correctness of object-oriented programs. [8]

3 Implementation in Rust

Rust's notion of ownership is the culmination of a long line of work. Many researchers have aimed to develop systems for functional programming without garbage collection. According to Weiss et al. [16], it is best understood that Rust's ownership model is based on Baker's work on Linear Lisp [5] where linearity enabled efficient reuse of objects in memory without garbage collection. Furthermore, the resemblance is especially strong between Rust *without* borrowing (the Rust terminology for referencing, explained in 3.5) and Baker's 'use-once' variables. 'Use-once' variables are bound to linear (unshared, unaliased or singly-referenced) objects and are cheap to access and manage because they require no synchronization or tracing garbage collection [6].

However, techniques like the 'use-once' variables differentiate themselves from Rust by requiring that all objects must be managed uniquely. Instead, Rust allows the use of shared references without mutation or unique references with unguarded mutation [16].

One can argue that Rust's ownership system is an extended version of the ownership model *owners-as-modifiers*. The owners-as-modifiers and Rust approaches differentiate themselves from the owners-as-dominators model, since they both allow read-only references to any object. Furthermore, as will be discussed in 3.5, Rust allows both read-only and mutable references in a controlled manner. This allows for expressing more programming idioms.

Most of the Rust-related information gathered in this chapter is derived from the "Understanding Ownership" section in the Rust Book [12].

3.1 Memory allocation system

We will begin our exploration of ownership types in Rust by defining its memory allocation system. There are two major requirements that a memory allocation system must fulfill: At runtime, memory can be requested from the memory allocator, and there exists a mechanism for memory to be freed by the memory allocator.

The first requirement is trivial.

One possible implementation of the second requirement is using a garbage collector, which keeps track of and cleans up memory that is no longer used. This, however, has a runtime performance overhead and can lead to a non-deterministic cleanup of resources. [11]

Another more traditional solution is requiring memory to be freed explicitly. This is a notorious source of bugs. If memory is never freed because it was forgotten, memory is wasted. If it is freed multiple times or accessed again after having been freed, it will result in undefined behavior.

The ideal solution pairs one *allocate* with exactly one *free*. Rust uses neither a garbage collector nor requires memory to be freed manually. Instead, the concept of ownership is leveraged to determine when to free memory.

The Rust Book [12] defines ownership as a set of rules that govern how it manages memory. The compiler checks whether these rules are adhered to and does not compile the program if they are violated. The ownership rules in Rust are as follows:

1. Each value in Rust has an owner.
2. There can only be one owner at a time.
3. When the owner goes out of scope, the value will be *dropped*.

The scope of a value is the range within a program in which it is valid. Rust automatically calls the function *drop* when a value goes out of scope, which frees the associated memory.

The following describes how the concept of ownership is implemented in Rust.

3.2 Transferring ownership

One common restriction to the early ownership systems is that the owner of an object must be set upon creation and then fixed for the lifetime of the object [8]. Rust extends these original ownership concepts by allowing the ownership of an object to be transferred.

First, one must differentiate between values that are stored on the stack and those stored on the heap. In Rust, primitive data types (e.g. signed 32-bit integers *i32*) are stored on the stack since their sizes are small and known at compile time. In contrast, values are stored on the heap if their size is dynamic and, therefore, unknown at compile time.

Heap-based values In addition to the memory safety that ownership types provide, ownership transfers can also reduce redundant copy operations. Since heap-based values could be arbitrarily large at runtime, copying the whole value instead of transferring ownership could be very inefficient. There are several different scenarios in which ownership is transferred.

In the simplest scenario, ownership is transferred from the variable *a* to the variable *b* when the variable *b* is assigned to the value of *a*. [1]

```rust
let a = HeapBasedType::default();
let b = a;
```

Listing 3. Rust simple heap-based move

In Rust terminology, *a* is *moved* into *b*. After the move, *a* is no longer valid, and *b* is the new owner of the value.

Heap-based values will always be generated by a function at runtime. In the above example, the return value of the function `HeapBasedType::default()` is moved into *a*. Furthermore, one can generalize the above scenario to ownership being transferred from the variable *a* to the variable *b* when *b* is assigned to the return value *a* of *foo*.

```rust
fn main() {
  let b = foo();
}
fn foo() {
  let a = HeapBasedType::default();
  a
}
```

Listing 4. Rust heap-based move out of function

Lastly, ownership is transferred from the variable *a* to the variable *b* when *a* is passed to function *foo*, which takes a parameter *b*.

```rust
1 fn main() {
2   let a = HeapBasedType::default();
3   foo(a);
4 }
5 fn foo(b: HeapBasedType) {
6   // do something with b
7 }
```

Listing 5. Rust heap-based move into function

In *foo*, *b* is the owner of the value until it is *dropped* at the end of the function. Again, accessing the moved-from variable after the ownership of that value has been transferred elsewhere will lead to a compile-time error. In this case, trying to access *a* after line 3 would not compile, as *a* is no longer the owner of the value.

The heap-allocated *String* type in Rust provides a good example of heap-based ownership transfer.

```rust
1 let x: String = String::from("hello world");
2 let y: String = x;
3 println!("x: {}, y: {}", x, y); // error
```

Listing 6. Rust String move

After *x* is moved into *y* in line 2, *x* is no longer valid. Therefore, Rust does not allow accessing the moved-from variable *x* in the third line, and the program will not compile.

[1]For the sake of simplicity, the main function is omitted in some code snippets

Stack-based values In contrast to heap-based values, the size of stack-based values is known at compile time and is usually small. Therefore, stack-based values in Rust are *copied* instead of *moved*. Internally, this works because primitive types are annotated with the *Copy* trait.

The following piece of code [12], which is similar to the code from above but replaces the *String* type with primitive integers, demonstrates how stack-based values are *copied* instead of *moved* into other variables in Rust.

```rust
let x: i32 = 42;
let y: i32 = x;
println!("x: {}, y: {}", x, y);
```

Listing 7. Rust stack-based move

The value of *x* is copied into *y*, after which both variables are still valid. Therefore, the program compiles and "x: 42, y: 42" is printed to stdout.

Moving fields out of structs In their work on the owners-as-modifiers property, Clarke et al. [8] emphasize that rather than simply protecting the fields of an object from external access, ownership types also protect the objects stored in the fields, thereby enabling an object to claim (exclusive) ownership of and access to other objects. This notion has been implemented in the Rust ownership system as well. The following example illustrates that not only whole objects, but also the fields of structures, can be moved into other objects.

```rust
1 struct Car {
2   engine: Engine,
3   wheels: Wheels,
4 }
5 fn main() {
6   let car = Car::default();
7   let moved_engine = car.engine; // move
8   car.drive() // error: use of partially moved
        value: 'car'
9 }
```

Listing 8. Rust moving struct fields

In line 7, we perform a so-called *partial move*, which means that parts of the variable *car* are moved into *engine*. The untouched parts, such as *wheels* in our example, will still be accessible afterward. After we move the *engine* field out of the *car* structure, however, the parent variable *car* cannot be used as a whole again. Therefore, the compiler will complain that we want to call the *drive* method on the partially moved value *car*.

3.3 The Clone trait

By default and by design, Rust will never automatically create deep copies of data stored on the heap. To do so, one must explicitly call the *clone* method on an object.

The following adjustment to the code snippet from above creates a deep copy of *x* and compiles successfully.

```rust
let x: String = String::from("hello world");
let y: String = x.clone();
println!("x: {}, y: {}", x, y);
```

Listing 9. Rust copy with clone

3.4 Immutability

By default, all values in Rust are immutable. An object is defined as immutable if its state cannot be modified after its creation. In Rust, mutable values and references must be explicitly created using the *mut* keyword.

For example, an integer can only be reassigned if it is mutable.

```rust
let a = 42;
a = 24; // compile error
let mut b = 42;
b = 24; // valid
```

Listing 10. Rust mutable variables

3.5 References and borrowing

Object aliasing has already been defined as the concept of accessing the same memory through different symbolic names. Specifically, the *owners-as-modifiers* approach relies on implementing the aliasing concept. Therefore, any programming language based on this form of ownership type must provide a way of achieving this behavior. One way of implementing this behavior is through *pointers*, which are commonly used in low-level programming languages. However, even though they exist, "working with raw pointers in Rust is uncommon [and] typically limited to a few patterns" [3]. Instead, Rust uses *references* as the preferred way of implementing aliasing.

In Rust, a *reference* is an address that points to a value that is owned by another variable. Unlike traditional *pointers*, a reference is guaranteed by the compiler to point to a valid value. In Rust terminology, creating a reference is called *borrowing*. A reference does not own the value it points to. Therefore, only the reference itself, not the value that the reference has borrowed, is dropped at the end of the reference's scope. Borrowing is useful for performing an operation on a value without taking ownership of it. Rust supports creating immutable and mutable references.

Lifetimes One detail we must discuss before providing details on immutable and mutable references is that every reference in Rust has a *lifetime* [12]. References borrow ownership and thus grant temporary access to a data structure [11]. A lifetime is a construct that the compiler, or more specifically, its *borrow checker*, uses to ensure that all borrows are valid [1]. The full form of a reference type, including lifetime annotations, is `&'a mut T` or `&'a T`, where `'a` is the lifetime of the reference. Our examples have not included explicit lifetime annotations until now because the compiler uses some conventions to elide them. To ensure references get used correctly, the compiler enforces the following two constraints [11]:

1. The reference can only be used while its lifetime is ongoing, and
2. the original referent is not used at all (for mutable references) or does not get mutated (for immutable references) until the lifetime of the newly created reference has expired.

The lifetime of a reference starts where it is introduced and extends until the last time it is used.

First, we will provide an example of a program that requires explicit lifetime annotations.

```
fn main() {
  let s1 = String::from("short");
  let s2 = String::from("longest");
  let longest = longest(&s1, &s2);
  println!("longest string: '{}'", longest);
}
fn longest<'a>(s1: &'a str, s2: &'a str) -> &'
    a str {
  if s1.len() > s2.len() {
    s1
  } else {
    s2
  }
}
```

Listing 11. Rust lifetime annotations [12]

If we omit the lifetime annotations `'a`, the compiler will tell us that the return type of the `longest` function contains a borrowed value, but it is unclear whether it is borrowed from *s1* or *s2*. In other words, the compiler requires that *s1* and *s2* have to live at least as long as lifetime `'a`.

Immutable references Immutable references can be seen as read-only references. They are acquired using the & operator and dereferenced using the * operator. The borrowed value that an immutable reference refers to may not be modified. Otherwise, the program will not compile.

```
let s = String::from("hello");
let r1 = &s; // 1. reference
let r2 = &s; // 2. reference
println!("r1: {}, r2: {}", *r1, *r2);
```

Listing 12. Rust immutable references

Furthermore, an unlimited amount of immutable references to the same value may exist at any time. The lifetimes of each of these immutable references may overlap, as demonstrated in the code example above. However, mutable references provide an exception to this rule, which is explained in the next section.

Multiple immutable references are allowed because none of the references can modify the value. Thereby, no immutable reference can affect another reference's reading of the value.

Mutable references Rust also supports mutable references, which are created with the `&mut` keyword, dereferenced with the * operator, and allow the borrowed value to be modified. For obvious reasons, creating a mutable reference is only possible if the borrowed value is also mutable.

```
let mut s = String::from("hello");
String::push_str(&mut s, " world");
// equivalent to
s.push_str(" world");
```

Listing 13. Rust mutable String reference

The lifetime of a mutable reference to a value may not overlap with the lifetime of any other immutable or mutable reference to that same value. This implies that there may only exist one mutable reference to a value at any given time. During this time, no further immutable or mutable references are allowed.

This restriction exists to identify and prevent *data races* at compile time, which can cause undefined behavior. A data race occurs when two or more pointers access the same data simultaneously, at least one of the pointers is being used to write the data, and there exists no mechanism to synchronize access to the data [12].

However, the following code example [12] shows that it is important to note that the creation of multiple mutable and immutable references with lifetimes that do not overlap is allowed.

```
1 let mut s = String::from("hello");
2
3 let r1 = &s; // allowed
4 let r2 = &s; // allowed
5 println!("{} and {}", r1, r2);
6
7 let r3 = &mut s; // allowed
8 r3.push_str(" world");
9 println!("{}", r3);
```

Listing 14. Rust non-overlapping lifetimes

Since *r1* and *r2* are not used after line 5, that is where their lifetimes end. The lifetime of the mutable reference *r3* only starts in line 7. Therefore, the lifetimes of the mutable reference and the two immutable references do not overlap, and the program compiles.

Lifetimes and mutable references also play an important role in explaining how the example from 1.1 got rejected by the compiler.

```
1 let mut v = vec![10, 11];
2 let vptr = &mut v[1]; // lifetime 'a starts
3 Vec::push(&mut v, 12); // lifetime 'b
4 println!("{}", *vptr); // lifetime 'a ends
```

Listing 15. Rust use-after-free with lifetimes [11]

The lifetime of the mutable reference *vptr* ranges from its creation in line 2 to its last use in line 4. Since *vptr* is a mutable reference to *v*, according to the second lifetime rule, *v* may not be used again during *vptr*'s lifetime. However, this condition is violated in line 3, when another reference to *v* is created, which implies that *v* is used. Therefore, the program does not compile.

By reordering lines 2 and 3, the lifetimes of the two mutable references will no longer overlap, allowing for a successful compilation of the program.

```
let mut v = vec![10, 11];
Vec::push(&mut v, 12); // lifetime 'b
let vptr = &mut v[1]; // lifetime 'a starts
println!("{}", *vptr); // lifetime 'a ends
```

Listing 16. Rust use-after-free corrected [11]

4 Evaluation

Given the breadth of research on ownership types explored in previous sections, it should not come as a surprise to learn that Rust is not the only programming language to use ownership types.

4.1 Other ownership implementations

In order to discuss and evaluate the advantages and disadvantages of Rust's ownership system, we will briefly compare its use of ownership to that of C++ using the work of Schroeder [14].

A first difference is that Rust included moving into the language's design early on, whereas C++ gained move semantics relatively late in its life.

C++ supports several different types of references based on "value categories" [2]. L-value references, denoted with a single ampersand, refer mainly to references to named values, while R-value references, denoted with two ampersands, refer mainly to temporary results of expression evaluations[2]. *Moving* in C++ is closely related to constructors and assignment operators, which are overloaded for each specific class to accept various types of references [14]. When invoking one of these special member functions, the *overload resolution process* will select the matching overload based on the value category of the reference. For L-values, the matching copy constructor or assignment operator will be called, while for R-values, the matching move constructor or assignment operator will be called. The following C++ example illustrates some significant differences between move semantics in C++ and Rust.

```
S s{};
S s_copied{s}; // copy constructor
S s_moved{std::move(s)}; // move constructor
// problem: 's' can still be used
```

Listing 17. C++ move semantics

We can construct a similar piece of code in Rust.

```
let s = S::default();
let s_copied = s.clone();
let s_moved = s; // 's' is moved
// compiler prevents use of 's'
```

Listing 18. Rust move semantics

Instead of using type-specific constructors and assignment operators, in Rust, we *clone* a value to copy it (stack-based primitives are copied automatically without an explicit clone) or directly *move* it without additional syntax. In C++, we must use `std::move` to move a named value, which casts it to an R-value reference,

[2]The provided definitions of L-values and R-values are oversimplifications but suffice for the purpose of this paper.

to indicate to the compiler that the move constructor for the specific type should be called [14]. After the move of *s* into *s_moved* in Rust, the compiler will not allow another use of the moved-from object *s*, as its lifetime has ended. In contrast, a `std::move` in C++ has no formal effect on the lifetime of the moved-from object *s*. This means that the object *s* can still be used after being moved, even though there is no guarantee by the compiler that it is still valid, because there exists no consistent specification on how move constructors or assignment operators should be implemented. We can conclude that Rust, in contrast to C++, adds safety to its move semantics through destructive moves enforced by the compiler [14].

We have only focused on the move semantics in the two languages, but there are many more ownership-related comparisons (smart pointers, shared ownership, mutable references etc.) between the two languages that are out of the scope of this paper.

5 Summary & Outlook

In summary, programming languages based on ownership types can eliminate many aliasing-related issues by enforcing strict rules on the existence and behavior of references to values. The *owners-as-dominators* and *owners-as-modifiers* models enforce different levels of strictness on both *where* and *how* references may be used. Furthermore, an enhanced version of the ownership concept is integral to the Rust programming language. Compared to more commonly used low-level programming languages like C++, Rust introduces many memory-related rules that are strictly enforced by the compiler at compile time. This allows for safer runtime performance and omits the need for many manual, rigorous and error-prone memory reference checks that programmers used to have to perform. Even though the Rust compiler forces developers to think more intentionally about how they deal with memory through ownership, they will receive helpful compile-time error messages instead of crashes or undefined behavior at runtime if they do something wrong.

It will be interesting to see how Rust's development will continue in the future. Given its rise in popularity as a safe and performant programming language using strict ownership types, the question arises of whether ownership types might also be incorporated into more new or existing programming languages in the future.

References

[1] 2022. *Rust by Example.* https://doc.rust-lang.org/rust-by-example/, accessed 2022-11-20.

[2] 2022. *Value categories.* https://en.cppreference.com/w/cpp/language/value_category, accessed 2023-01-21.

[3] 2023. *Rust pointer type documentation.* https://doc.rust-lang.org/std/primitive.pointer.html, accessed 2023-01-15.

[4] 2023. *Rust vector type documentation.* https://doc.rust-lang.org/std/vec/struct.Vec.html, accessed 2023-01-15.

[5] Henry G. Baker. 1992. *Lively linear Lisp: "look ma, no garbage!".*

[6] Henry G. Baker. 1995. *'Use-Once' Variables and Linear Objects - Storage Management, Reflection and Multi-Threading.*

[7] Microsoft Security Response Center. 2019. *Trends, challenges, and strategic shifts in the software vulnerability mitigation landscape.* https://github.com/microsoft/MSRC-Security-Research/tree/master/presentations/2019_02_BlueHatIL, accessed 2022-12-04.

[8] Dave Clarke, Johan Östlund, Ilya Sergey, and Tobias Wrigstad. 2013. *Ownership Types: A Survey.*

[9] David G. Clarke, John M. Potter, and James Noble. 1998. *Ownership Types for Flexible Alias Protection.* https://doi.org/10.1145/286936.286947

[10] Werner Dietl and Peter Müller. 2005. Universes: Lightweight Ownership for JML. *J. Object Technology* 4 (2005), 5–32.

[11] Ralf Jung. 2020. *Understanding and evolving the Rust programming language.*

[12] Steve Klabnik and Carol Nichols. 2019. *The Rust Programming Language.* https://doc.rust-lang.org/book/, accessed 2022-11-19.

[13] NSA Media Relations. 2022. *Software Memory Safety.* https://media.defense.gov/2022/Nov/10/2003112742/-1/-1/0/CSI_SOFTWARE_MEMORY_SAFETY.PDF, accessed 2022-11-19.

[14] Paul Schroeder. 2022. C++ & Rust: (Interior) Mutability, Moving and Ownership. (june 2022). https://www.tangramvision.com/blog/c-rust-interior-mutability-moving-and-ownership, accessed 2023-01-21.

[15] Adrian Taylor, Andrew Whalley, Dana Jansens, and Nasko Oskov. 2021. *An update on Memory Safety in Chrome.* https://security.googleblog.com/2021/09/an-update-on-memory-safety-in-chrome.html, accessed 2022-12-04.

[16] Aaron Weiss, Daniel Patterson, Nicholas D. Matsakis, and Amal Ahmed. 2019. Oxide: The Essence of Rust. *CoRR* abs/1903.00982 (2019).

[17] Yang Zhao and John Boyland. 2008. *A Fundamental Permission Interpretation for Ownership Types.* https://doi.org/10.1109/TASE.2008.45