Safe Systems Programming in Rust: The Promise and the Challenge

Ralf Jung
MPI-SWS, Germany

Jacques-Henri Jourdan
Université Paris-Saclay, CNRS, LRI, France

Robbert Krebbers
TU Delft, The Netherlands

Derek Dreyer
MPI-SWS, Germany

Key Insights

- Rust is the first industry-supported programming language to overcome the longstanding tradeoff between the *safety* guarantees of higher-level languages (like Java) and the *control* over resource management provided by lower-level "systems programming" languages (like C and C++).
- It tackles this challenge using a *strong type system* based on the ideas of *ownership and borrowing*, which statically prohibits the mutation of shared state. This approach enables many common systems programming pitfalls to be detected at compile time.
- There are a number of data types whose implementations fundamentally depend on shared mutable state and thus cannot be typechecked according to Rust's strict ownership discipline. To support such data types, Rust embraces the judicious use of *unsafe code encapsulated within safe APIs*.
- The proof technique of *semantic type soundness*, together with advances in *separation logic* and *machine-checked proof*, has enabled us to begin building rigorous formal foundations for Rust as part of the RustBelt project.

There is a longstanding tension in programming language design between two seemingly irreconcilable desiderata.

- **Safety**. We want strong type systems that rule out large classes of bugs statically. We want automatic memory management. We want data encapsulation, so that we can enforce invariants on the private representations of objects and be sure that they will not be broken by untrusted code.
- **Control**. At least for "systems programming" applications like web browsers, operating systems, or game engines, where performance or resource constraints are a primary concern, we want to determine the byte-level representation of data. We want to optimize the time and space usage of our programs using low-level programming techniques. We want access to the "bare metal" when we need it.

Sadly, the conventional wisdom goes, we can't have everything we want. Languages like Java give us strong safety, but it comes at the expense of control. As a result, for many systems programming applications, the only realistic option is to use a language like C or C++ that provides fine-grained control over resource management. However, this control comes at a steep cost. For example, Microsoft recently reported that 70% of the security vulnerabilities they fix are due to memory safety violations [33], precisely the type of bugs that strong type systems were designed to rule out. Likewise, Mozilla reports that the vast majority of critical bugs they find in Firefox are memory-related [16]. If only there were a way to somehow get the best of both worlds: a *safe systems programming language with control*...

Enter **Rust**. Sponsored by Mozilla and developed actively over the past decade by a large and diverse community of contributors, Rust supports many common low-level programming idioms and APIs derived from modern C++. However, unlike C++, Rust enforces the safe usage of these APIs with a strong static type system.

In particular, like Java, Rust protects programmers from memory safety violations (*e.g.*, "use-after-free" bugs). But Rust goes further by defending programmers against other, more insidious anomalies that no other mainstream language can prevent. For example, consider *data races*: unsynchronized accesses to shared memory (at least one of which is a write). Even though data races effectively constitute undefined (or weakly-defined) behavior for concurrent code, most "safe" languages (such as Java and Go) permit them, and they are a reliable source of concurrency bugs [35]. In contrast, Rust's type system rules out data races at compile time.

Rust has been steadily gaining in popularity, to the point that it is now being used internally by many major industrial software vendors (such as Dropbox, Facebook, Amazon, and Cloudflare) and has topped StackOverflow's list of "most loved" programming languages for the past four years. Microsoft's Security Response Center Team recently announced that it is actively exploring an investment in the use of Rust at Microsoft to stem the tide of security vulnerabilities in system software [25, 8].

The design of Rust draws deeply from the wellspring of academic research on safe systems programming. In particular, the most distinctive feature of Rust's design—in relation to other mainstream languages—is its adoption of an *ownership type system* (which in the academic literature is often referred to as an *affine* or *substructural* type system [36]). Ownership type systems help the programmer enforce safe patterns of lower-level programming by placing restrictions on which *aliases* (references) to an object may be used to mutate it at any given point in the program's execution.

However, Rust goes beyond the ownership type systems of prior work in at least two novel and exciting ways:

1. Rust employs the mechanisms of *borrowing* and *lifetimes*, which make it much easier to express common C++-style idioms and ensure that they are used safely.
2. Rust also provides *a rich set of APIs—e.g.*, for concurrency abstractions, efficient data structures, and memory management—which fundamentally extend the power of the language by supporting more flexible combinations of aliasing and mutation than Rust's core type system allows. Correspondingly, these APIs cannot be implemented within the safe fragment of Rust: rather, they internally make use of potentially *unsafe* C-style features of the language, but in a *safely encapsulated* way that is claimed not to disturb Rust's language-level safety guarantees.

These aspects of Rust's design are not only essential to its success—they also pose fundamental research questions about its semantics and safety guarantees that the programming languages community is just beginning to explore.

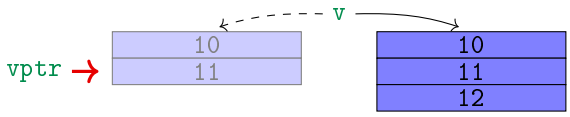
In this article, we begin by giving the reader a bird's-eye view of the Rust programming language, with an emphasis on some of the essential features of Rust that set it apart from its contemporaries. Second, we describe the initial progress made in the RustBelt project, an ongoing project funded by the European Research Council (ERC), whose goal is to provide the first formal (and machine-checked) foundations for the safety claims of Rust. In so doing, we hope to inspire other members of the computer science research community to start paying closer attention to Rust and to help contribute to the development of this groundbreaking language.

Motivation: Pointer Invalidation in C++

To demonstrate the kind of memory safety problems that arise commonly in systems programming languages, let us consider the following C++ code:

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

In the first line, this program creates a `std::vector` (a growable array) of integers. The initial contents of `v`, the two elements `10` and `11`, are stored in a buffer in memory. In the second line, we create a pointer `vptr` that points *into* this buffer; specifically it points 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*. In the third line, we push a new element to the end of `v`. The element `12` is added after `11` in the buffer backing `v`. If there is no more space for an additional element, a new buffer is allocated and all the existing elements are moved over. Let us assume this is what happens here. Why is this case 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. This is possible because both pointers were aliasing: an action through a pointer (`v`) will in general also affect all its aliases (`vptr`). The entire situation is visualized as follows:



The fact that `vptr` is now a dangling pointer becomes a problem in the fourth line. Here we load from `vptr`, and since it is a dangling pointer, this is a use-after-free bug.

In fact, the problem is common enough that one instance of it has its own name: *iterator invalidation*, which refers to the situation where an iterator (usually internally implemented with a pointer) gets invalidated because the data structure it iterates over is mutated during the iteration. It most commonly arises when one iterates over some container data structure in a loop, and indirectly, but accidentally, calls an operation that mutates the data structure. Notice that in practice the call to the operation that mutates the data structure (`push_back` in line 3 of our example) might be deeply nested behind several layers of abstraction. In particular when code gets refactored or new features get added, it is often near impossible to determine if pushing to a certain vector will invalidate pointers elsewhere in the program that are going to be used again later.

Comparison with garbage-collected languages

Languages like Java, Go, and OCaml avoid use-after-free bugs using garbage collection: memory is only deallocated when it can no longer be used by the program. Thus, there can be no dangling pointers and no use-after-free.

One problem with garbage collection is that, to make it efficient, such languages generally do not permit *interior* pointers (*i.e.*, pointers *into* data structures). For example, arrays `int[]` in Java are represented similarly to `std::vector<int>` in C++ (except arrays in Java cannot be grown). However, unlike in C++, one can only *get* and *set* elements of a Java array, not take *references* to them. To make the elements themselves addressable, they need to be separate objects, references to which can then be stored in the array—*i.e.*, the elements need to be "boxed". This sacrifices performance and control over memory layout in return for safety.

On top of that, garbage collection does not even properly solve the issue of iterator invalidation. Mutating a collection while iterating over it in Java cannot lead to dangling pointers, but it may lead to a `ConcurrentModificationException` being thrown at run time. Similarly, while Java *does* prevent security vulnerabilities caused by null pointer misuse, it does so with run-time checks that raise a `NullPointerException`. In both of these cases, while the result is clearly better than the corresponding undefined behavior of a C++ program, it still leaves a lot to be desired: instead of shipping incorrect code and then detecting issues at run time, we want to prevent the bugs from occurring in the first place.

Rust's solution to pointer invalidation

In Rust, issues like iterator invalidation and null pointer misuse are detected statically, by the compiler—they lead to a compile-time error instead of a run-time exception. To explain how this works, 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
```

Like in the C++ version, there is a buffer in memory, and `vptr` points into the middle of that buffer (causing aliasing); `push` might reallocate the buffer, which leads to `vptr` becoming a dangling pointer, and that leads to a use-after-free in line 4.

But none of this happens; instead the compiler shows an error: "cannot borrow `v` as mutable more than once at a time". We will come back to "borrowing" soon, but the key idea—the mechanism through which Rust achieves memory safety in the presence of pointers that point into a data structure—already becomes visible here: the type system enforces the discipline (with a notable exception that we will come to later) that *a reference is never both aliased and mutable at the same time*. This principle should sound familiar in the context of concurrency, and indeed Rust uses it to ensure the absence of data races as well. However, as our example that is rejected by the Rust compiler shows, the unrestricted combination of aliasing and mutation is a recipe for disaster even for sequential programs: in line 3, `vptr` and `v` alias (`v` is considered to point to all of its contents, which overlaps with `vptr`), and we are performing a mutation, which would lead to a memory access bug in line 4.

Ownership and Borrowing

The core mechanism through which Rust prevents uncontrolled aliasing is *ownership*. Memory in Rust always has a unique owner, as demonstrated by the following example:

```
1  fn consume(w: Vec<i32>) {
2    drop(w); // deallocate vector
3  }
4  let v = vec![10, 11];
5  consume(v);
6  v.push(12); // Compiler error
```

Here, we construct `v` similar to our first example, and then pass it to `consume`. Operationally, just like in C++, parameters are passed by value but the copy is shallow—pointers get copied but their pointee does not get duplicated. This means that `v` and `w` point to the same buffer in memory.

Such aliasing is a problem if `v` and `w` would *both* be used by the program, but an attempt to do so in line 6 leads to a compile-time error. This is because Rust considers ownership of `v` to have *moved* to `consume` as part of the call, meaning that `consume` can do whatever it desires with `w`, and the caller may no longer access the memory backing this vector at all.

Resource management. Ownership in Rust not only prevents memory bugs—it also forms the core of Rust's approach to memory management and, more generally, resource management. When a variable holding owned memory (*e.g.*, a variable of type `Vec<T>`, which owns the buffer in memory backing the vector) goes out of scope, we know for sure that this memory will not be needed any more—so the compiler can automatically deallocate the memory at that point. To this end, the compiler transparently inserts *destructor* calls, just like in C++. For example, in the `consume` function, it is not actually necessary to call the destructor method (`drop`) explicitly. We could have just left the body of that function empty, and it would have automatically deallocated `w` itself.

As a consequence, Rust programmers rarely have to worry about memory management: it is largely automatic, despite the lack of a garbage collector. Moreover, the fact that memory management is also *static* (determined at compile time) yields enormous benefits: it helps not only to keep the maximal memory consumption down, but also to provide good *worst-case* latency in a reactive system such as a web server. And on top of that, Rust's approach generalizes beyond memory management: other resources like file descriptors, sockets, lock handles, and so on are handled with the same mechanism, so that Rust programmers do not have to worry, for instance, about closing files or releasing locks. Using destructors for automatic resource management was pioneered in the form of RAII (Resource Acquisition Is Initialization) in C++ [31]; the key difference in Rust is that the type system can statically ensure that resources do not get used any more after they have been destructed.

Mutable references. A strict ownership discipline is nice and simple, but unfortunately not very convenient to work with. Frequently, one wants to provide data to some function *temporarily*, but get it back when that function is done. For example, we want `v.push(12)` to grant `push` the privilege to mutate `v`, but we do not want it to *consume* the vector `v`.

In Rust, this is achieved through *borrowing*, which takes a lot of inspiration from prior work on *region types* [34, 13]. For example, we could write:

```
1  fn add_something(v: &mut Vec<i32>) {
2    v.push(11);
3  }
4  let mut v = vec![10];
5  add_something(&mut v);
6  v.push(12); // Ok!
7  // v.push(12) is syntactic sugar for
8  // Vec::push(&mut v, 12)
```

The function `add_something` takes an argument of type `&mut Vec<i32>`, which indicates a *mutable reference* to a `Vec<i32>`. Operationally, this acts just like a reference in C++, *i.e.*, the `Vec` is passed by reference. In the type system, this is interpreted as `add_something` *borrowing* ownership of the `Vec` from the caller.

The function `add_something` demonstrates what borrowing looks like in well-formed programs. To see why the compiler accepts that code while rejecting our pointer invalidation example from page 3, we have to introduce another concept: *lifetimes*. Just like in real life, when borrowing something, misunderstanding can be prevented by agreeing up front on how long something may be borrowed. So, when a reference gets created, it gets assigned a lifetime, which gets recorded in the full form of the reference type: `&'a mut T` for lifetime `'a`. The compiler ensures that (a) the reference (`v`, in our example) only gets used during that lifetime, and (b) the referent does not get used again until the lifetime is over.

In our case, the lifetimes (which are all inferred by the compiler) just last for the duration of `add_something` and `Vec::push`, respectively. Never is `v` used while the lifetime of a previous borrow is still ongoing.

In contrast, consider the example from page 3:

```
1 let mut v = vec![10, 11];
2 let vptr : &'a mut i32 = &mut v[1];
3 v.push(12);
4 println!("{}", *vptr); // Compiler error
                                  Lifetime 'a
```

The lifetime `'a` of the borrow for `vptr` starts in line 2 and goes on until line 4. It cannot be any shorter because `vptr` gets used in line 4. However, this means that in line 3, `v` is used while an outstanding borrow exists, which is an error.

To summarize: whenever something is passed *by value* (as in `consume`), Rust interprets this as *ownership transfer*; when something is passed *by reference* (as in `add_something`), Rust interprets this as *borrowing* for a certain *lifetime*.

Shared references. Following the principle that we can have either aliasing or mutation, but not both at the same time, mutable references are unique pointers: they do not permit aliasing. To complete this picture, Rust has a second kind of reference, the *shared reference* written `&Vec<i32>` or `&'a Vec<i32>`, which allows aliasing but no mutation. One primary use-case for shared references is to share read-only data between multiple threads:

```
1 let v = vec![10,11];
2 let vptr = &v[1];
3 join( || println!("v[1] = {}", *vptr),
4       || println!("v[1] = {}", *vptr));
5 v.push(12);
```

Here, we create a shared reference `vptr` pointing to (and borrowing) `v[1]`. The vertical bars here represent a *closure* (also sometimes called an anonymous function or "lambda") that does not take any arguments. These closures are passed to `join`, which is the Rust version of "parallel composition": it takes two closures, runs both of them in parallel, waits until both are done, and returns both of their results. When `join` returns, the borrow ends, so we can mutate `v` again.

Just like mutable references, shared references have a lifetime. Under the hood, the Rust compiler is using a lifetime to track the period during which `v` is temporarily shared between the two threads; after that lifetime is over (on line 7), the original owner of `v` regains full control. The key difference here is that multiple shared references are allowed to coexist during the same lifetime, so long as they are only used for *reading*, not writing. We can witness the enforcement of this restriction by changing one of the two threads in our example to `|| v.push(12)`: then the compiler complains that we cannot have a mutable reference and a shared reference to the `Vec` at the same time. And indeed, that program has a fatal data race between the reading thread and the thread that pushes to the vector, so it is important that the compiler detects such cases statically.

Shared references are also useful in sequential code; for example, while doing a shared iteration over a vector we can still pass a shared reference to the *entire* vector to another function. But for this article, we will focus on the use of sharing for concurrency.

Summary

In order to obtain safety, the Rust type system enforces the discipline that a reference is never both aliased and mutable. Having a value of type `T` means you "own" it fully. The value of type `T` can be "borrowed" using a mutable reference (`&mut T`) or shared reference (`&T`).

Relaxing Rust's Strict Ownership Discipline via Safe APIs

Rust's core ownership discipline is sufficiently flexible to account for many low-level programming idioms. But for implementing certain data structures, it can be overly restrictive. For example, without any mutation of aliased state, it is not possible to implement a doubly-linked list because each node is aliased by both its next and previous neighbors.

Rust adopts a somewhat unusual approach to this problem. Rather than either (1) complicating its type system to account for data structure implementations that do not adhere to it, or (2) introducing dynamic checks to enforce safety at run time, Rust allows its ownership discipline to be relaxed through the development of *safe APIs*—APIs that extend the expressive power of the language by enabling safely controlled usage of aliased mutable state. Although the implementations of these APIs do not adhere to Rust's strict ownership discipline (a point we return to on page 6), the APIs themselves make critical use of Rust's ownership and borrowing mechanisms to ensure that they preserve the safety guarantees of Rust as a whole. Let us now look at a few examples.

Shared mutable state

Rust's shared references permit multiple threads to read shared data concurrently. But threads that just *read* data are only half the story, so next we will look at how the `Mutex`

API enables one to safely share *mutable* state across thread boundaries. At first, this might seem to contradict everything we said so far about the safety of Rust: isn't the whole point of Rust's ownership discipline that it *prevents* mutation of shared state? Indeed it is, but we will see how, using `Mutex`, such mutation can be sufficiently restricted so as to not break memory or thread safety. Consider the following example:

```
1  let mutex_v = Mutex::new(vec![10, 11]);
2  join(
3    || { let mut v = mutex_v.lock().unwrap();
4         v.push(12); },
5    || { let v = mutex_v.lock().unwrap();
6         println!("{:?}", *v) });
```

We again use structured concurrency and shared references, but now we wrap the vector in a `Mutex`: the variable `mutex_v` has type `Mutex<Vec<i32>>`. The key operation on a `Mutex` is `lock`, which blocks until it can acquire the exclusive lock. The lock implicitly gets released by `v`'s destructor when the variable goes out of scope. Ultimately, this program prints either `[10, 11, 12]` if the first thread manages to acquire the lock first, or `[10, 11]` if the second thread does.

In order to understand how our example program type-checks, let us take a closer look at `lock`. It (almost[1]) has type `fn(&'a Mutex<T>) -> MutexGuard<'a, T>`. This type says that `lock` can be called with a shared reference to a mutex, which is why Rust lets us call `lock` on both threads: both closures capture an `&Mutex<Vec<i32>>`, and as with the `vptr` of type `&i32` that got captured in our first concurrency example, both threads can then use that reference concurrently. In fact, it is crucial that `lock` take a shared rather than a mutable reference—otherwise, two threads could not attempt to acquire the lock at the same time and there would be no need for a lock in the first place.

The return type of `lock`, namely `MutexGuard<'a, T>`, is basically the same as `&'a mut T`: it grants exclusive access to the `T` that is stored inside the lock. Moreover, when it goes out of scope, it automatically releases the lock (an idiom known in the C++ world as RAII [31]).

In our example, this means that both threads *temporarily* have exclusive access to the vector, and they have a mutable reference that reflects that fact—but thanks to the lock properly implementing mutual exclusion, they will never both have a mutable reference *at the same time*, so the uniqueness property of mutable references is maintained. In other words, `Mutex` can offer mutation of aliased state safely because it implements run-time checks ensuring that, *during* each mutation, the state is *not* aliased.

Reference counting

We have seen that shared references provide a way to share data between different parties in a program. However, shared references come with a *statically determined* lifetime, and when that lifetime is over, the data is uniquely owned again. This works well with structured parallelism (like `join` in the previous example), but does not work with *unstructured* parallelism where threads are spawned off and keep running independently from the parent thread.

In Rust, the typical way to share data in such a situation is to use an *atomically reference-counted* pointer: `Arc<T>` is a pointer to `T`, but it also counts how many such pointers exist and deallocates the `T` (and releases its associated resources) when the last pointer is destroyed. (This can be viewed as a form of lightweight library-implemented garbage collection.) Since the data is shared, we cannot obtain an `&mut T` from an `Arc<T>`—but we *can* obtain an `&T` (where the compiler ensures that during the lifetime of the reference, the `Arc<T>` does not get destroyed), as in this example:

```
1  let arc_v1 = Arc::new(vec![10, 11]);
2  let arc_v2 = Arc::clone(&arc_v1);
3  spawn(move || println!("{:?}", arc_v2[1]));
4  println!("{:?}", arc_v1[1]);
```

We start by creating an `Arc` that points to our usual vector. `arc_v2` is obtained by *cloning* `arc_v1`, which means that the reference count gets bumped up by one, but the data itself is not duplicated. Then we `spawn` a thread that uses `arc_v2`; this thread keeps running in the background even when the function we are writing here returns. Because this is unstructured parallelism we have to explicitly `move` (*i.e.*, transfer ownership of) `arc_v2` into the closure that runs in the other thread. `Arc` is a "smart pointer" (similar to `shared_ptr` in C++), so we can work with it almost as if it were an `&Vec<i32>`. In particular, in lines 3 and 4 we can use indexing to print the element at position 1. Implicitly, as `arc_v1` and `arc_v2` go out of scope, their destructors get called, and the last `Arc` to be destroyed deallocates the vector.

Thread safety

There is one last type that we would like to talk about in this brief introduction to Rust: `Rc<T>` is a reference-counted type very similar to `Arc<T>`, but with the key distinction that `Arc<T>` uses an *atomic* (fetch-and-add) instruction to update the reference count, whereas `Rc<T>` uses *non-atomic* memory operations. As a result, `Rc<T>` is potentially faster, but *not* thread-safe. The type `Rc<T>` is useful in complex sequential code where the static scoping enforced by shared references is not flexible enough, or where one cannot statically predict when the last reference to an object will be destroyed so that the object itself can be deallocated.

Since `Rc<T>` is not thread-safe, we need to make sure that the programmer does not accidentally use `Rc<T>` when they should have used `Arc<T>`. This is important: if we take our previous `Arc` example, and replace all the `Arc` by `Rc`, the program has a data race and might deallocate the memory too early or not at all. However, quite remarkably, the Rust compiler is able to catch this mistake. The way this works is

[1] The actual type of `lock` wraps the result in a `LockResult<...>` for error handling, which explains why we use `unwrap` on lines 3 and 5.

that Rust employs something called the `Send` *trait*: a property of types which is only enjoyed by a type `T` if elements of type `T` can be safely sent to another thread. The type `Arc<T>` is `Send`, but `Rc<T>` is not. Both `join` and `spawn` require everything captured by the closure(s) they run to be `Send`, so if we capture a value of the non-`Send` type `Rc<T>` in a closure, compilation will fail.

Rust's use of the `Send` trait demonstrates how sometimes the restrictions imposed by strong static typing can lead to *greater* expressive power, not less. In particular, C++'s smart reference-counted pointer, `std::shared_ptr`, always uses atomic instructions[2], because having a more efficient non-thread-safe variant like `Rc` is considered too risky. In contrast, Rust's `Send` trait allows one to "hack without fear" [26]: it provides a way to have both thread-safe data structures (such as `Arc`) *and* non-thread-safe data structures (such as `Rc`) in the same language, while ensuring modularly that the two do not get used in incorrect ways.

Unsafe Code, Safely Encapsulated

We have seen how types like `Arc` and `Mutex` let Rust programs safely use features such as reference counting and shared mutable state. However, there is a catch: *those types cannot actually be implemented in Rust*. Or, rather, they cannot be implemented in *safe* Rust: the compiler would reject an implementation of `Arc` for potentially violating the aliasing discipline. In fact, it would even reject the implementation of `Vec` for accessing potentially uninitialized memory. For efficiency reasons, `Vec` manually manages the underlying buffer and tracks which parts of it are initialized. Of course, the implementation of `Arc` does *not* in fact violate the aliasing discipline, and `Vec` does *not* in fact access uninitialized memory, but the arguments needed to establish those facts are too subtle for the Rust compiler to infer.

To solve this problem, Rust has an "escape hatch": Rust consists not only of the safe language we discussed so far—it also provides some *unsafe* features such as C-style unrestricted pointers. The safety (memory safety and/or thread safety) of these features cannot be guaranteed by the compiler, so they are only available inside syntactic blocks that are marked with the `unsafe` keyword. This way, one can be sure to not *accidentally* leave the realm of safe Rust.

For example, the implementation of `Arc` uses unsafe code to implement a pattern that would not be expressible in safe Rust: sharing without a clear owner, managed by thread-safe reference counting. This is further complicated by support for "weak references": references that do not keep the referent alive, but can be atomically checked for liveness and upgraded to a full `Arc`. The correctness of `Arc` relies on rather subtle concurrent reasoning, and the Rust compiler simply has no way to verify statically that deallocating the memory when the reference count reaches zero is in fact safe.

[2] More precisely, on Linux it uses atomic instructions if the program uses `pthreads`, *i.e.*, if it or any library it uses *might* spawn a thread.

Alternatives to `unsafe` blocks. One could turn things like `Arc` or `Vec` into language primitives. For example, Python and Swift have built-in reference counting, and Python has `list` as a built-in equivalent to `Vec`. However, these language features are implemented in C or C++, so they are not actually any safer than the unsafe Rust implementation. Beyond that, restricting unsafe operations to implementations of language primitives also severely restricts flexibility. For example, Firefox uses a Rust library implementing a variant of `Arc` without support for weak references, which improves space usage and performance for code that does not need them. Should the language provide primitives for every conceivable spot in the design space of any built-in type?

Another option to avoid unsafe code is to make the type system expressive enough to actually be able to verify safety of types like `Arc`. However, due to how subtle correctness of such data structures can be (and indeed `Arc` and simplified variants of it have been used as a major case-study in several recent formal verification papers [12, 18, 9]), this basically requires a form of general-purpose theorem prover—and a researcher with enough background to use it. The theorem proving community is quite far away from enabling developers to carry out such proofs themselves.

Safe abstractions. Rust has instead opted to allow programmers the flexibility of writing unsafe code when necessary, albeit with the expectation that it should be *encapsulated by safe APIs*. Safe encapsulation means that, regardless of the fact that Rust APIs like `Arc` or `Vec` are implemented with unsafe code, *users* of those APIs should not be affected: so long as users write well-typed code in the safe fragment of Rust, they should never be able to observe anomalous behaviors due to the use of unsafe code in the APIs' implementation. This is in marked contrast to C++, whose weak type system lacks the ability to even enforce that APIs are *used* safely. As a result, C++ APIs like `shared_ptr` or `vector` are prone to misuse, leading to reference-counting bugs and iterator invalidation, which do not arise in Rust.

The ability to write unsafe code is like a lever that Rust programmers use to make the type system more useful without turning it into a theorem prover, and indeed we believe this to be a key ingredient to Rust's success. The Rust community is developing an entire ecosystem of safely usable high-performance libraries, enabling programmers to build safe and efficient applications on top of them.

But of course, there is no free lunch: it is up to the author of a Rust library to somehow ensure that, if they write unsafe code, they are being very careful not to break Rust's safety guarantees. On the one hand, this is a much better situation than in C/C++, because the vast majority of Rust code is written in the safe fragment of the language, so Rust's "attack surface" is much smaller. On the other hand, when unsafe code is needed, it is far from obvious how a programmer is supposed to know if they are being "careful" enough.

To maintain confidence in the safety of the Rust ecosystem, we therefore really want to have a way of formally specifying and verifying what it means for uses of unsafe code to be safely encapsulated behind a safe API. This is precisely the goal of the **RustBelt** project.

RustBelt: Securing the Foundations of Rust

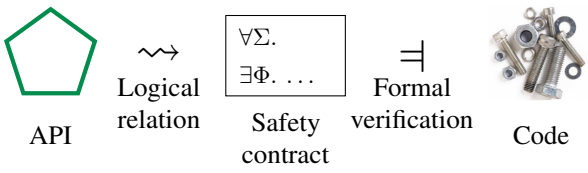
The key challenge in verifying Rust's safety claims is accounting for the interaction between safe and unsafe code. To see why this is challenging, let us briefly take a look at the standard technique for verifying safety of programming languages—the so called *syntactic approach* [37, 14]. Using that technique, safety is expressed in terms of a *syntactic typing judgment*, which gives a formal account of the type checker in terms of a number of mathematical inference rules.

Theorem 1 (Syntactic type soundness)**.** *If a program e is well-typed w.r.t. the syntactic typing judgment, then e is safe.*

Unfortunately, this theorem is too weak for our purposes, because it only talks about *syntactically* safe programs, thus ruling out programs that use unsafe code. For example, `if true { e } else { crash() }` is not syntactically well-typed, but it is still safe since `crash()` is never executed.

The key solution: Semantic type soundness

To account for the interaction between safe and unsafe code, we instead use a technique called *semantic type soundness*, which expresses safety in terms of the "behavior" of the program rather than a fixed set of inference rules. The key ingredient of semantic soundness is a *logical relation*, which assigns a *safety contract* to each API. It expresses that if the inputs to each method in the API conform to their specified types, then so do the outputs. Using techniques from formal verification, one can then prove that an implementation of the API satisfies the assigned safety contract:



Semantic type soundness is ideal for reasoning about programs that use a combination of safe and unsafe code. For any library that uses unsafe code (such as `Arc`, `Mutex`, `Rc`, and `Vec`) one has to prove by hand that the implementation satisfies the safety contract. For example:

Theorem 2. `Arc` *satisfies its safety contract.*

For safe pieces of a program, the verification is automatic. This is expressed by the following theorem, which says that if a component is written in the safe fragment of Rust, it satisfies its safety contract by construction.

Theorem 3 (Fundamental theorem)**.** *If a component e is syntactically well-typed, then e satisfies its safety contract.*

Together, these imply that a Rust program is safe if the only appearances of `unsafe` blocks are within libraries that have been manually verified to satisfy their safety contracts.

Using the Iris logic to encode safety contracts

Semantic type soundness is an old technique, dating back at least to Milner's seminal 1978 paper on type soundness [28], but scaling it up to realistic modern languages like Rust has proven a difficult challenge. In fact, scaling it up to languages with mutable state and higher-order functions remained an open problem until the development of "step-indexed Kripke logical relations" (SKLR) models [5, 3] as part of the Foundational Proof-Carrying Code project [4, 2] in the early 2000s. Even then, verifications of safety contracts that were encoded directly using SKLR models turned out to be very tedious, low-level, and difficult to maintain.

In RustBelt we build upon more recent work on Iris [21, 19, 23, 20], a verification framework for higher-order, concurrent, imperative programs, implemented in the Coq proof assistant [1]. Iris provides a much higher-level language for encoding and working with SKLR models, thus enabling us to scale such models to handle a language as sophisticated as Rust. In particular, Iris is based on *separation logic* [29, 30], an extension of Hoare logic [15] geared specifically toward modular reasoning about pointer-manipulating programs, and centered around the concept of ownership. This provides us with an ideal language in which to model the semantics of ownership types in Rust.

Iris extends traditional separation logic with several additional features that are crucial for modeling Rust:

- Iris supports *user-defined ghost state*: the ability to define custom logical resources that are useful for proving correctness of a program but do not correspond directly to anything in its physical state. Iris's user-defined ghost state has enabled us to verify the soundness of libraries like `Arc`, for which ownership does not correspond to physical ownership (*e.g.*, two separately-owned `Arc<T>`'s may be backed by the same underlying memory)—a phenomenon known as "fictional separation" [11, 10]. It has also enabled us to reason about Rust's borrowing and lifetimes at a much higher level of abstraction, by deriving (within Iris) a new, domain-specific "lifetime logic".
- Iris supports *impredicative invariants*: invariants on the program state that may refer cyclically to the existence of other invariants [32]. Impredicative invariants play an essential role in modeling central type system features such as recursive types and closures.

The complexity of Rust demands that our semantic soundness proofs be *machine-checked*, as it would be too tedious and error-prone to do proofs by hand. Fortunately, Iris comes with a rich set of *separation-logic tactics*, which are patterned after standard Coq tactics and thus make it possible to interactively develop machine-checked semantic soundness proofs in a time-tested style familiar to Coq users [24, 22].

Conclusion and Outlook

In this article we have given a bird's-eye view of Rust, demonstrating its core concepts like borrowing, lifetimes, and unsafe code encapsulated inside safe APIs. These features have helped Rust become the first industry-supported language to overcome the tradeoff between safety and control.

To formally investigate Rust's safety claims, we described the proof technique of semantic type soundness, which has enabled us to begin building a rigorous foundation for Rust in the RustBelt project. For more details about Rust and RustBelt, we refer the interested reader to our POPL'18 paper [18] and the first author's forthcoming PhD thesis [17].

There is still much work left to do. Although RustBelt has recently been extended to account for the relaxed-memory concurrency model that Rust inherits from C++ [9], there are a number of other Rust features and APIs that it does not yet cover, such as its "trait" system, which is complex enough to have been the source of subtle soundness bugs [7]. Moreover, although verifying the soundness of an internally-unsafe Rust library requires, at present, a deep background in formal semantics, we hope to eventually develop formal methods that can be put directly in the hands of programmers.

Finally, while RustBelt has focused on building foundations for Rust itself, we are pleased to see other research projects (notably Prusti [6] and RustHorn [27]) beginning to explore an exciting, orthogonal direction: namely, the potential for Rust's strong type system to serve as a powerful tool in simplifying the formal verification of systems code.

References

[1] The Coq proof assistant, 2019. https://coq.inria.fr/.

[2] A. Ahmed, A. W. Appel, C. D. Richards, K. N. Swadi, G. Tan, and D. C. Wang. Semantic foundations for typed assembly languages. *TOPLAS*, 32(3), 2010.

[3] A. J. Ahmed. *Semantics of types for mutable state*. PhD thesis, Princeton University, 2004.

[4] A. W. Appel. Foundational proof-carrying code. In *LICS*, 2001.

[5] A. W. Appel and D. McAllester. An indexed model of recursive types for foundational proof-carrying code. *TOPLAS*, 23(5), 2001.

[6] V. Astrauskas, P. Müller, F. Poli, and A. J. Summers. Leveraging Rust types for modular specification and verification. *PACMPL*, 3(OOPSLA), 2019.

[7] A. Ben-Yehuda. Coherence can be bypassed by an indirect impl for a trait object, 2019. https://github.com/rust-lang/rust/issues/57893.

[8] A. Burch. Using Rust in Windows, 2019. Blog post. https://msrc-blog.microsoft.com/2019/11/07/using-rust-in-windows/.

[9] H.-H. Dang, J.-H. Jourdan, J.-O. Kaiser, and D. Dreyer. RustBelt meets relaxed memory. *PACMPL*, 4(POPL), 2020.

[10] T. Dinsdale-Young, M. Dodds, P. Gardner, M. J. Parkinson, and V. Vafeiadis. Concurrent abstract predicates. In *ECOOP*, 2010.

[11] T. Dinsdale-Young, P. Gardner, and M. J. Wheelhouse. Abstraction and refinement for local reasoning. In *VSTTE*, 2010.

[12] M. Doko and V. Vafeiadis. Tackling real-life relaxed concurrency with FSL++. In *ESOP*, volume 10201 of *LNCS*, 2017.

[13] D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney. Region-based memory management in Cyclone. In *PLDI*, 2002.

[14] R. Harper. *Practical Foundations for Programming Languages (2nd Edition)*. Cambridge University Press, 2016.

[15] C. A. R. Hoare. An axiomatic basis for computer programming. *CACM*, 12(10), 1969.

[16] D. Hosfelt. Implications of rewriting a browser component in Rust, 2019. Blog post. https://hacks.mozilla.org/2019/02/rewriting-a-browser-component-in-rust/.

[17] R. Jung. *Understanding and Evolving the Rust Programming Language*. PhD thesis, Universität des Saarlandes, 2020 (expected). https://plv.mpi-sws.org/rustbelt/jung-thesis/.

[18] R. Jung, J.-H. Jourdan, R. Krebbers, and D. Dreyer. RustBelt: Securing the foundations of the Rust programming language. *PACMPL*, 2(POPL), 2018.

[19] R. Jung, R. Krebbers, L. Birkedal, and D. Dreyer. Higher-order ghost state. In *ICFP*, 2016.

[20] R. Jung, R. Krebbers, J.-H. Jourdan, A. Bizjak, L. Birkedal, and D. Dreyer. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *JFP*, 28, 2018.

[21] R. Jung, D. Swasey, F. Sieczkowski, K. Svendsen, A. Turon, L. Birkedal, and D. Dreyer. Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning. In *POPL*, 2015.

[22] R. Krebbers, J.-H. Jourdan, R. Jung, J. Tassarotti, J.-O. Kaiser, A. Timany, A. Charguéraud, and D. Dreyer. MoSeL: A general, extensible modal framework for interactive proofs in separation logic. *PACMPL*, 2(ICFP), 2018.

[23] R. Krebbers, R. Jung, A. Bizjak, J. Jourdan, D. Dreyer, and L. Birkedal. The essence of higher-order concurrent separation logic. In *ESOP*, 2017.

[24] R. Krebbers, A. Timany, and L. Birkedal. Interactive proofs in higher-order concurrent separation logic. In *POPL*, 2017.

[25] R. Levick. Why Rust for safe systems programming, 2019. Blog post. https://msrc-blog.microsoft.com/2019/07/22/why-rust-for-safe-systems-programming/.

[26] N. Matsakis and A. Turon. Rust in 2016, 2015. Blog post. https://blog.rust-lang.org/2015/08/14/Next-year.html.

[27] Y. Matsushita, T. Tsukada, and N. Kobayashi. RustHorn: CHC-based verification for Rust programs. In *ESOP*, 2020.

[28] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3), 1978.

[29] P. W. O'Hearn, J. C. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *CSL*, 2001.

[30] P. W. O'Hearn. Resources, concurrency, and local reasoning. *Theoretical Computer Science*, 375(1-3), 2007.

[31] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, 2013.

[32] K. Svendsen and L. Birkedal. Impredicative concurrent abstract predicates. In *ESOP*, 2014.

[33] G. Thomas. A proactive approach to more secure code, 2019. Blog post. https://msrc-blog.microsoft.com/2019/07/16/a-proactive-approach-to-more-secure-code/.

[34] M. Tofte and J. Talpin. Region-based memory management. *Information and Computation*, 132(2), 1997.

[35] T. Tu, X. Liu, L. Song, and Y. Zhang. Understanding real-world concurrency bugs in Go. In *ASPLOS*, 2019.

[36] D. Walker. Substructural type systems. In B. C. Pierce, editor, *Advanced Topics in Types and Programming Languages*. MIT Press, 2005.

[37] A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and computation*, 115(1), 1994.