

# **Safety in an unsafe { world }**

**Joshua Liebow-Feeser**

Thank you so much to the conference organizers and to the other speakers. This has been a really amazing event.

Now we've got a lot to get through, so let's jump in.

**“Computer science is not the study of how to produce correct programs. It is the study of how **humans** can produce correct programs.”**

**Aristotle**

## About me

- Fuchsia security @ Google
- Adjunct professor @ University of San Francisco (Intro to Computer Security)
- Netstack3
- zerocopy
- Mundane
- Founded the Rust Secure Code Working Group

Contact info  
& references



# Netstack3

So what is this talk about? Well I'm going to bury the lede a bit - I'll tell you in a few slides.

First, let's talk about Netstack3. Netstack3 is Fuchsia's next-generation, pure-Rust networking stack. It aims to replace Netstack2, which is written in Go.

I started Netstack3 six years ago, and led its development for four years. Many engineers have worked on Netstack3, and of course I haven't even been on the project for the past two years, so this is absolutely a team effort.

Now, writing a networking stack is a serious undertaking. It's responsible for almost all network traffic in the entire operating system. It implements dozens of protocols, each of which is specified in documents which can run into the hundreds of pages. And it's the first line of defense for any attacker who isn't physically sitting in front of a device.

It's also just... big.

6 years

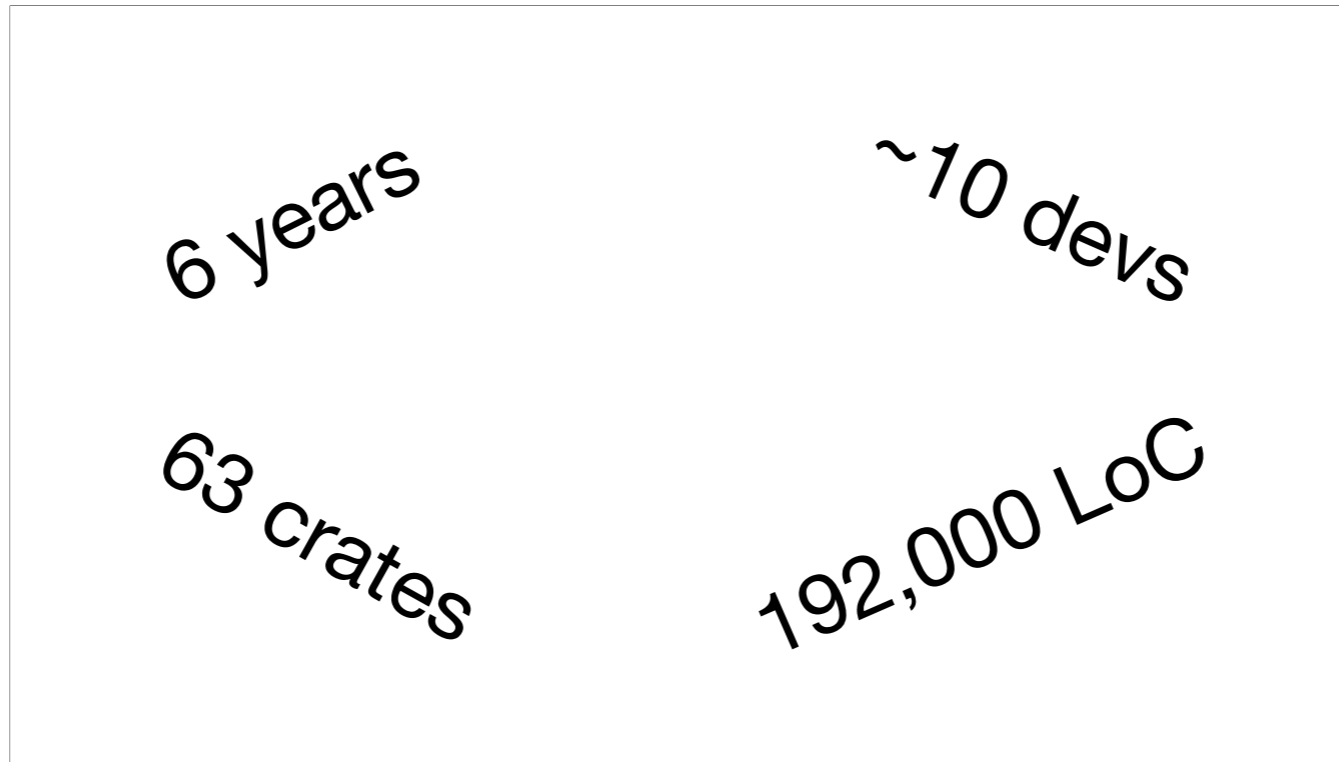
6 years

~10 devs

6 years

~10 devs

63 crates



That's larger than the top 10 crates on [crates.io](https://crates.io) combined.

So over the past year, the team has been preparing to launch Netstack3. Those of you with networking backgrounds will know...





...that you don't simply deploy networking code to production. Networking code is famously difficult to test, and so you have to assume that, despite your best efforts, your code is riddled with bugs.

For a project of this scale - deploying an entirely new, ground-up rewrite of a netstack - you would expect to dogfood in the field for months or years before shipping to real users. In that time, you would expect to find and fix tens to hundreds of bugs that hadn't shown up during development.

Once you had burned down most of these bugs and seen relatively stable behavior over an extended period of time, only then would you finally deploy to real users.

So how has Netstack3 done?

## Netstack3 Dogfood

- 11 months
- ~60 devices
- On ~24/7
- Bugs identified in the field: ???

Well, for the past 11 months, the team has been slowly ramping up a dogfooding program. At peak, that program has seen about 60 devices running nearly 24/7 in developers' homes.

Again, if this were any other netstack, we'd have expected to uncover mountains of bugs in that time. So how many bugs did the team find?

## Netstack3 Dogfood

- 11 months
- ~60 devices
- On ~24/7
- Bugs identified in the field: 3

Three.



This talk is going to be something of a flashback.

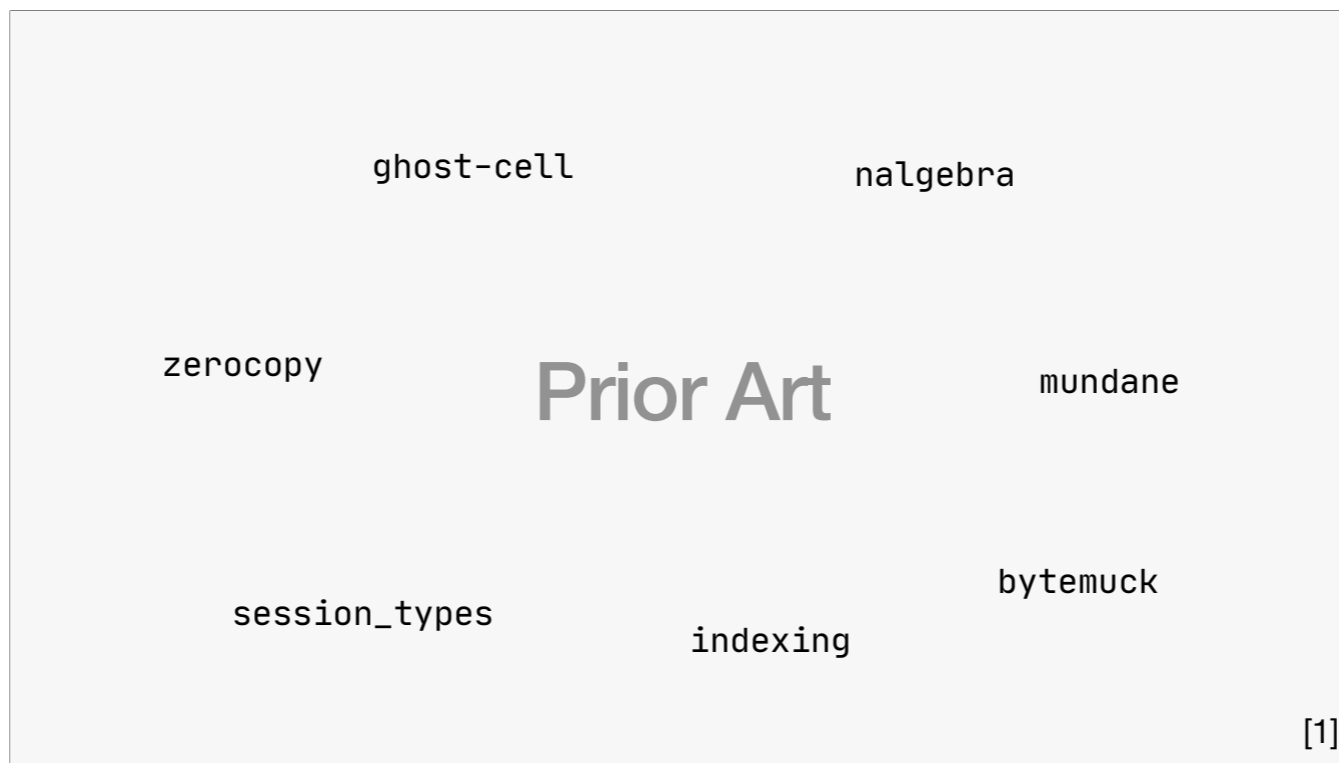


That's better.

Netstack3 was designed around a particular methodology for how to architect robust systems. Hopefully I've convinced you that we did at least *something* right on the robustness front.

**Buggy programs don't compile**

In a sentence, that methodology says: Design your abstractions so that buggy programs don't compile.



Now, obviously I am far from the first person to suggest this methodology. Here is just a small sample of crates that employ this methodology somewhere in their APIs.

References are like jumps  
*withoutboats*

Some notes on Rust, mutable aliasing and formal verification  
*Graydon Hoare*

Type Safety Back and Forth  
*Matt Parsons*

Parse, don't validate  
*Alexis King*

# Prior Art

Ghosts of Departed Proofs  
*Matt Noonan*

Compiler-Driven Development in Rust  
*No Boilerplate*

The Typestate Pattern in Rust  
*Cliff Biffle*

[2]

And here is a small sample of what's been written on this subject, both about Rust and about other languages.



# Goals

So my goal for this talk is not to introduce a new idea.

Instead, my goals are the following:

First, I'm going to propose a concrete but general framework that attempts to unify all of the different ways that this methodology has been applied in practice, and explains them in terms of the same basic concepts.

Second, I'm going to walk through two examples - one from the standard library, and one from Netstack3 - and show how we can explain them in terms of this framework.

Unfortunately, 25 minutes isn't enough time to present more than two examples in the depth that I'd like, and I had to leave a *lot* of examples on the cutting room floor, including the original research that I mention in the talk abstract - sorry about that. But I promise that these two examples only scratch the surface of what's possible. I have a long list of references that I'll mention at the end of the talk, and I strongly encourage anyone who's interested to read through them. I think you'll be surprised by the diversity of problems that this methodology has been applied to. I also personally have many more examples from Netstack3 and from other projects, so if you want to learn more about those, please come talk to me or ask on Discord.

## Why Rust?

### Performance

Rust is blazingly fast and memory-efficient: with no runtime or garbage collector, it can power performance-critical services, run on embedded devices, and easily integrate with other languages.

### Reliability

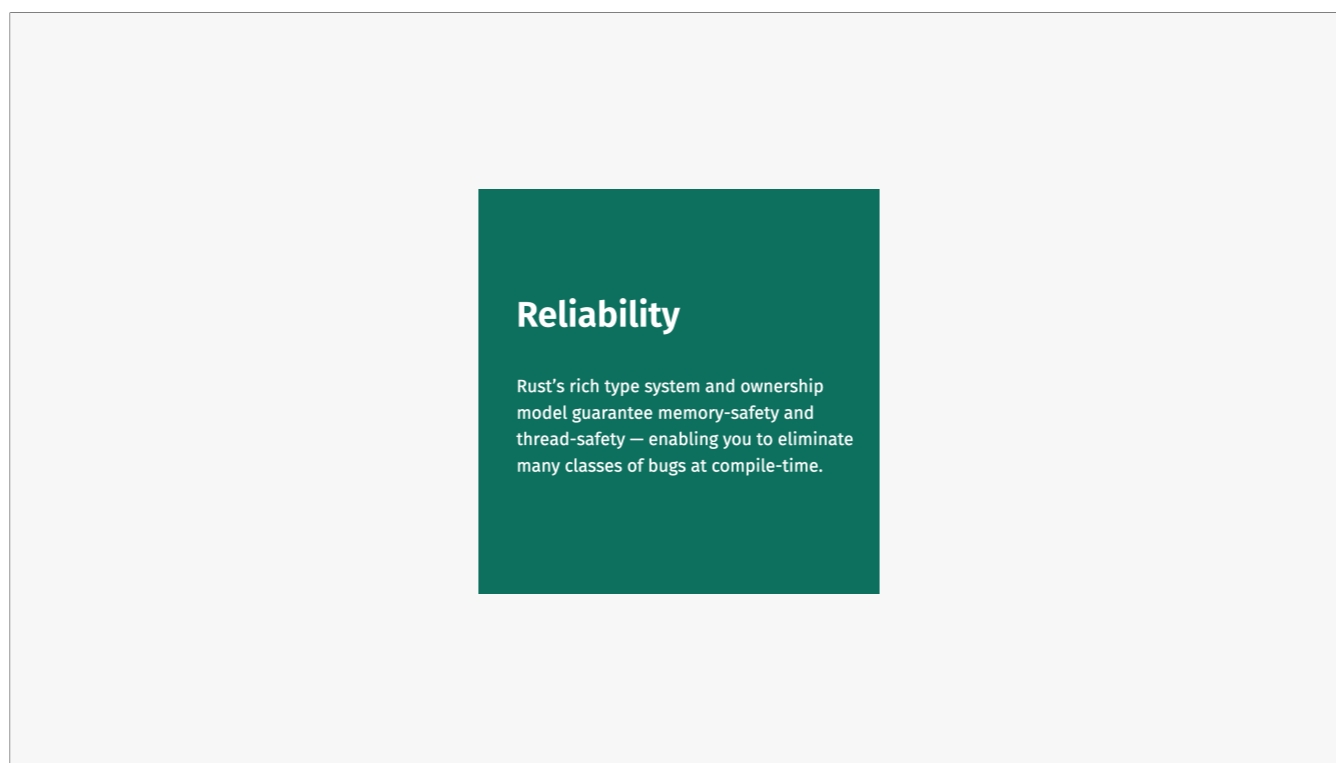
Rust's rich type system and ownership model guarantee memory-safety and thread-safety — enabling you to eliminate many classes of bugs at compile-time.

### Productivity

Rust has great documentation, a friendly compiler with useful error messages, and top-notch tooling — an integrated package manager and build tool, smart multi-editor support with auto-completion and type inspections, an auto-formatter, and more.

So what exactly do we mean by “buggy programs don’t compile”?

On its website, Rust advertises three properties: performance, reliability, and productivity. Let’s zoom in on reliability.



The website says: “Rust’s rich type system and ownership model guarantee memory-safety and thread-safety.”

In other words, out of the box, Rust guarantees that “buggy programs don’t compile”... but only if we restrict our definition of “bug” to memory bugs or threading bugs.

Now don’t get me wrong - memory- and thread- safety are huge on their own. For example, Netstack3 is able to pull all sorts of crazy buffer sharing tricks in the name of performance that would be wildly dangerous in C or C++. I gave a talk on those techniques at Rust Belt Rust in 2018 if you’re curious about the details. There will be a link in the references at the end. But that’s only half the story.

Many critical bugs that we’d like to prevent are neither memory bugs nor threading bugs.

# Basic Functionality

RFC 4614 Section 2

[3]

Take TCP for example. If you want to implement what RFC 4614 refers to as “basic functionality” for TCP, you need to implement six different standards which run a combined 270 pages.

# Recommended Enhancements

RFC 4614 Section 3

[4]

Add in the “recommended enhancements” and you’re up to a total of 18 standards spanning 476 pages.

And that’s just one protocol. Add Ethernet, ARP, NDP, IPv4, IPv6, ICMP, IGMP, MLD, UDP, and a host of others, and all the subtle interactions between them, and you start to realize that memory and threading bugs are just the tip of the iceberg of all the fun and exciting ways that you could fuck it up.

That’s why uncovering only three bugs in almost a year of dogfooding is so surprising.

There’s just no way that memory- or thread-safety alone would get you that level of robustness.

# Rust is a **memory**-safe language

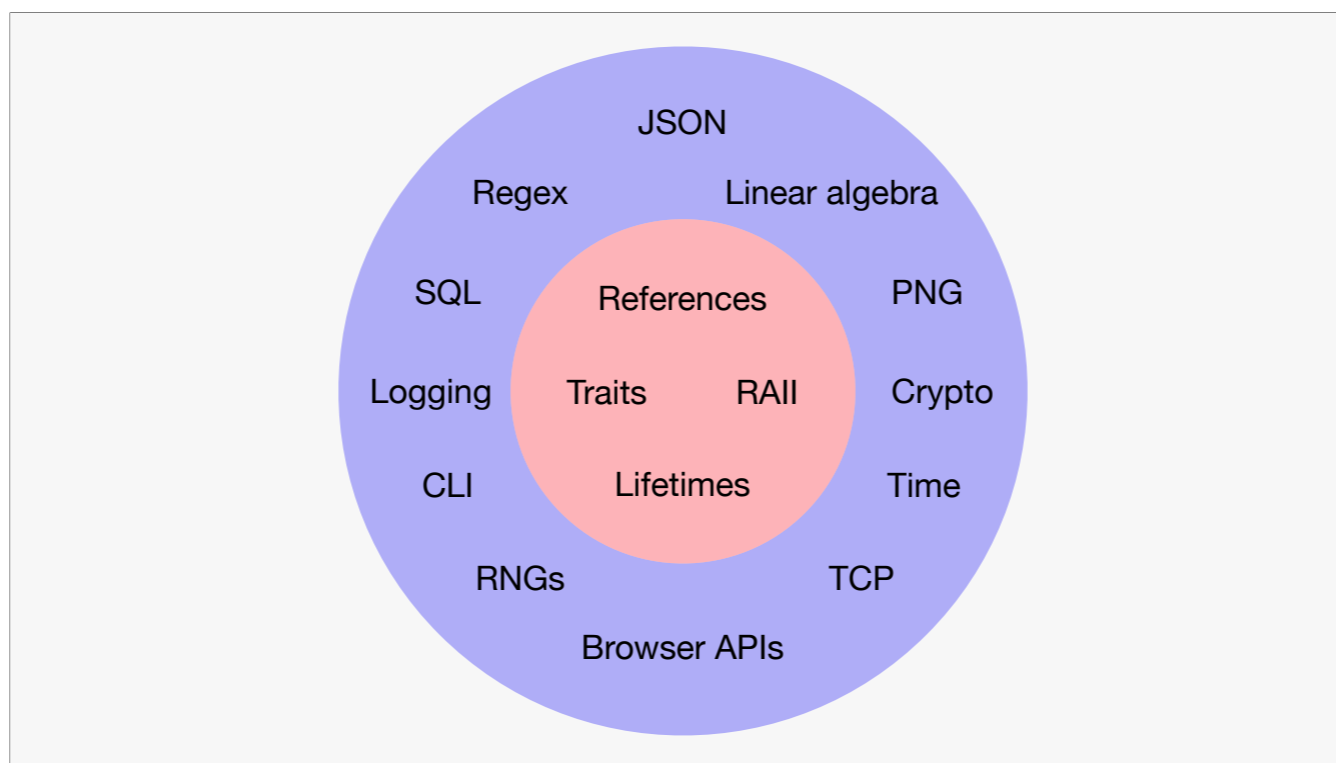
So the points that I'm trying to make are:

One: That, out of the box, Rust guarantees that “buggy programs don't compile” only with respect to a small subset of the bugs we care about, and that,

Two: Netstack3 would not have been able to achieve the level of robustness that it has by relying only on what's provided by the language out-of-the-box.

## Rust is an **X**-safe language

However, what Netstack3 has done, and what I believe that *any* project can do, is extend the Rust language to guarantee freedom from almost any class of bugs - not just memory or threading bugs. This is what I mean by “X-safety” in the abstract for this talk.



As it stands today, Rust provides a core set of abstractions, and guarantees various safety properties related to those abstractions. But the majority of abstractions in the ecosystem are provided by libraries, which generally don't provide the same degree of safety with respect to their abstractions compared to what we might expect from the language.

But it doesn't have to be that way. As we'll see in the following examples, Rust's core abstractions alone are sufficient to express almost any safety property. I claim that every library can and should provide the same level of safety with respect to its abstractions that we already expect of the language itself.



# **A general framework for safety**

So let's get into the framework.

```
struct Node<T> {  
    left: Option<Box<Node<T>>>,  
    right: Option<Box<Node<T>>>,  
    value: T,  
}
```

To explain it, we're going to turn to the humble binary tree.

In order for our binary tree to be correct, we of course need memory safety, but we need another safety property as well: we have to ensure that the tree is *ordered* - that every value in the left sub-tree is less than our node's value, and that our node's value is less than every value in the right sub-tree.

Rust can of course guarantee memory safety, but it has no visibility into this ordering requirement, and so it can't enforce it for us. So how do we make sure that we never produce an invalidly-ordered tree?

```
struct Node<T> {  
    // INVARIANT: All values in `left` are less than `value`.  
    left: Option<Box<Node<T>>>,  
    // INVARIANT: All values in `right` are greater than `value`.  
    right: Option<Box<Node<T>>>,  
    value: T,  
}
```

We use the concept of an *invariant*.

First, we document an ordering invariant that we expect to always hold for any **Node**.

```
struct Node<T> {
    // INVARIANT: All values in `left` are less than `value`.
    left: Option<Box<Node<T>>>,
    // INVARIANT: All values in `right` are greater than `value`.
    right: Option<Box<Node<T>>>,
    value: T,
}

impl<T> Node<T> {
    // Post-condition: Node's internal field invariants hold.
    fn new(value: T) → Node<T> {
        Node { left: None, right: None, value }
    }
}

}
```

Next, we ensure that every time we create a new **Node**, the ordering invariant holds.

```

struct Node<T> {
    // INVARIANT: All values in `left` are less than `value`.
    left: Option<Box<Node<T>>>,
    // INVARIANT: All values in `right` are greater than `value`.
    right: Option<Box<Node<T>>>,
    value: T,
}

impl<T> Node<T> {
    // Post-condition: Node's internal field invariants hold.
    fn new(value: T) → Node<T> {
        Node { left: None, right: None, value }
    }

    // Pre-condition: Node's internal field invariants hold.
    // Post-condition: Node's internal field invariants hold.
    fn insert(&mut self, value: T) → Option<T> { ... }
}

```

And finally, we ensure that every time we modify an existing **Node**, we preserve the ordering invariant, assuming it held to begin with.

Now imagine now that you are code *outside* of this module (and we'll restrict ourself here to safe code - obviously unsafe code can do whatever it wants). I claim that, from the perspective of safe code outside of this module, there is *no difference whatsoever* between safety invariants provided by the language and the ordering invariant provided by this module. If you write code which, if run, would violate Rust's memory safety guarantees, then Rust guarantees that that code will not compile. By a similar token, if you write code *outside of this module* which, if run, would violate the ordering invariant, then Rust guarantees that that code will not compile.

In other words, by structuring our code in this way, we have in some sense *taught Rust about a new safety property that it didn't know about before*.

# Framework

We can use this **Node** example to explain our framework. It has three components.

## Definition

```
struct Node<T>
```

Ordering property

First, “definition.” We define an object which the Rust type system can reason about. In this case, we define the **Node** struct. We attach to this object a safety property which Rust *cannot* reason about. In this case, the ordering property.

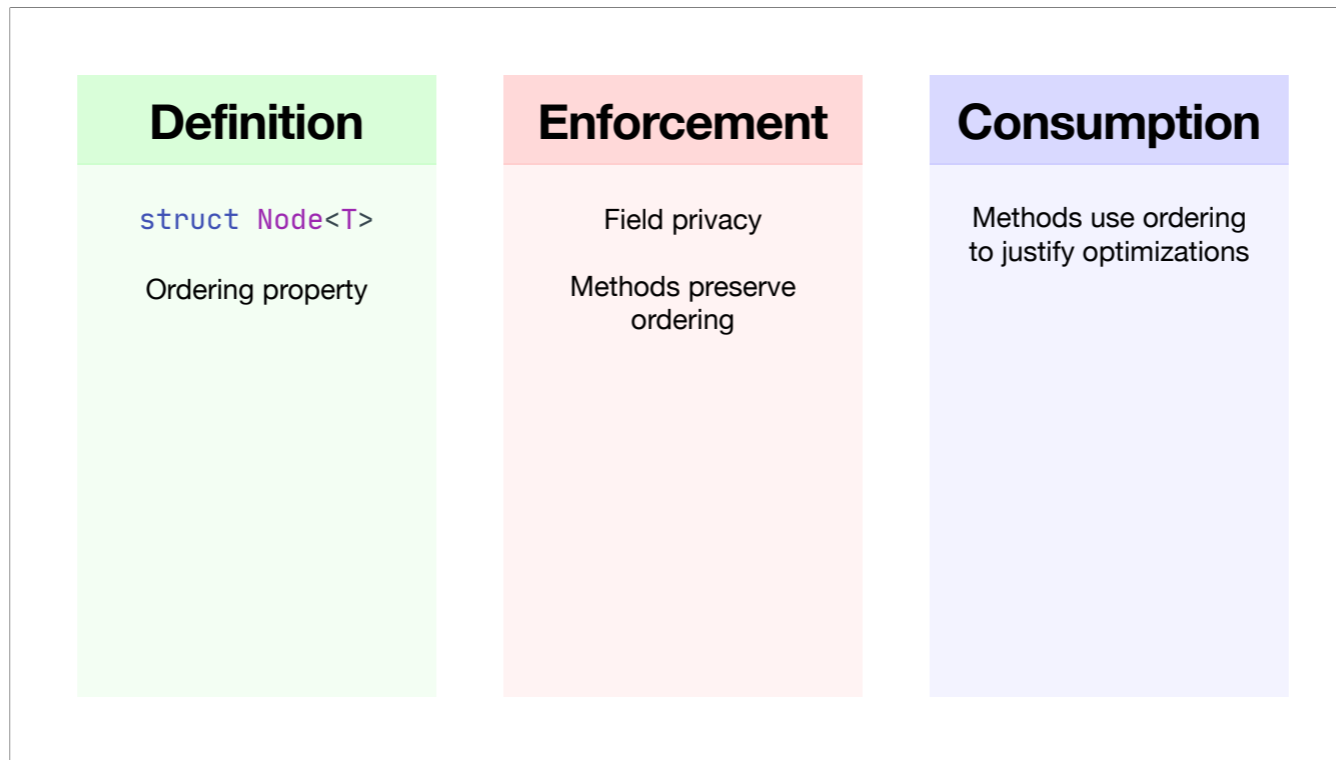
Because Rust can’t reason about this safety property, it’s up to us as the authors of the abstraction to make sure to never violate it.



This brings us to the second component, which is “enforcement.” Here, we enforce that our safety property is upheld.

First, we make sure that our struct fields are private. If they were public, then external code could modify them and violate our safety property. Second, we make sure that all of the code we write preserves the safety property. In this case, the methods which modify the tree are responsible for maintaining the order.





Finally, the third component is “consumption.” This is where we get the bang for our buck. We write code which consumes the safety property as a pre-condition, and is only correct in virtue of that safety property being upheld. In this case, our methods can find the correct location in the tree for a particular value in  $\log(N)$  time, without having to traverse the entire tree. That optimization is only valid because we are guaranteed that the tree is ordered, and the tree is only guaranteed to be ordered because we have structured our code to *define* and *enforce* this ordering property so that we can later consume it as a precondition.

So with that framework in mind, let’s get to our first example.

# Thread Safety

Our first example is from the standard library and concerns thread safety.

What I'm going to demonstrate with this example is that not only can libraries use the language's core features to build new safety guarantees, but that the standard library itself already uses this technique! In fact, the thread safety that Rust advertises as one of its core features is not actually a language feature at all - it's implemented in the standard library, and in principle could have been implemented in a third-party crate. It's just convenient to have it in the standard library.

```
fn spawn<F: FnOnce()>(f: F)
```

So let's imagine that you're writing **spawn**. `spawn` is a real function from the `thread` module in the standard library, although I've stripped down its signature for simplicity. It takes a function, and spawns a new thread which will run that function.

If we think about how we might implement **spawn**, we immediately run into a limitation in the language - the Rust language itself doesn't actually provide any mechanism to interact with threads. Threads are an operating system concept, and so creating a new thread requires going outside the language and making use of an API provided by the operating system itself.

```
fn spawn<F: FnOnce()>(f: F) {  
    unsafe { libc::pthread_create(...); }  
}
```

For example, on a POSIX system, this might require calling libc's **pthread\_create** function. This is where we run into the first “core abstraction” that we'll need to use as a building block. Since libc functions like **pthread\_create** are implemented outside of the language, Rust has no visibility into their behavior, and so it has no way of guaranteeing that calling such a function won't violate memory safety.

This is why Rust introduces the notion of **unsafe** code. **Unsafe** code is code which Rust can't guarantee *won't* violate memory safety. That doesn't mean that it definitely *will* violate memory safety. It just means that, on its own, Rust isn't smart enough to prove that it won't. Instead, Rust has to rely on the programmer to do the heavy lifting and confirm that the **unsafe** code is correct - or “sound”.

This is the purpose of an **unsafe** block like the one in this slide. Inside of an **unsafe** block, Rust gives the programmer free rein to perform operations that Rust can't reason about. In exchange, the programmer takes responsibility for upholding memory safety.

This is our first example of using a core abstraction to extend the power of the language. Rust itself provides **unsafe** functions and **unsafe** blocks, but it doesn't provide any mechanism for interacting with threads. However, by using **unsafe** in combination with external APIs like **pthread\_create**, we can “teach” the language about threads like we've done here.

—

Now, as written, this implementation of **spawn** is unsound. Not all values are “thread-safe”, meaning that not all values are sound to send between threads - think of the difference between **Arc** and **Rc**. As written, there's nothing to stop you from passing a closure which captures some value which is not thread-safe.

```
/// # Safety
///
/// `Self` is thread-safe.
unsafe trait Send {}
```

So what the standard library does is encode the notion of thread safety in the type system.

First, we introduce a new trait called **Send**. This is *the* standard library **Send** that you all know and love.

Note that **Send** is an **unsafe** trait. Just like **unsafe** functions, **unsafe** traits tell the type system: “this trait has safety implications which you can’t reason about.” Just as calling an **unsafe** function can only happen inside an **unsafe** block, implementing an **unsafe** trait requires an **unsafe** impl block. Just as before, the programmer must opt into taking on the responsibility for upholding memory safety.

Since the language can’t reason about the meaning of **Send**, we have to document its meaning in prose. In this case, we say that a type can only be **Send** if it’s thread-safe.

```
/// # Safety
///
/// `Self` is thread-safe.
unsafe trait Send {}

fn spawn<F: FnOnce() + Send>(f: F) {
    unsafe { libc::pthread_create(...) };
}
```

Now that we've introduced this **Send** trait, we can use it as a bound elsewhere in our program. In this case, by requiring that **F** implement **Send**, we can make **spawn** sound. There's no longer a risk that a programmer could accidentally call **spawn** with a non-thread-safe value because that wouldn't type check.

—

So we've introduced this **Send** trait that encodes the notion of thread-safety, but we haven't implemented it for anything. Let's do that.

```
/// # Safety
///
/// `Self` is thread-safe.
unsafe trait Send {}

unsafe impl Send for u8 {}
unsafe impl Send for u16 {}
unsafe impl Send for u32 {}
```

First, we can implement **Send** for any built-in type which we know is always thread-safe.

From Rust's perspective, this is axiomatic. Using these **unsafe impl** blocks, we've declared by fiat that the **Send** property holds of these types. Again, Rust has no way of knowing what **Send** means, and so it has no way of knowing whether these impls are correct - it's just taking our word for it.

```
/// # Safety
///
/// `Self` is thread-safe.
unsafe trait Send {}

unsafe impl<'a, T: Send> Send
    for &'a Mutex<T>
{
}
```

We can also implement **Send** for more complex types. For example, a mutex is thread-safe by design, and so it's okay to send mutex references between threads.



```
#[derive(Send)]  
struct Foo<T, U>(T, U);
```

Finally, we can permit arbitrary composite types to implement **Send** by writing a custom derive. Of course, in reality, **Send** uses a special mechanism known as “auto traits” to accomplish the same thing, but my point in this slide is to reiterate that there’s no reason in principle that **Send** couldn’t have just been developed as a library, maintained by somebody other than the Rust project. Again, the core abstractions provided by the language are enough on their own to permit somebody to build an abstraction like **Send** outside of the standard library.

```
#[derive(Send)]
struct Foo<T, U>(T, U);

// Derive-generated code
unsafe impl<T, U> Send for Foo<T, U>
where
    T: Send,
    U: Send,
{
}
```

Sticking with the hypothetical, our custom derive would automatically emit an **unsafe impl** block with the appropriate bounds. In particular, a type can be **Send** if all of its fields are **Send**.

So let's tie this all together.

```
#[derive(Send)]
struct Foo<T, U>(T, U);

let foo: Foo<&Mutex<u8>, u16> = ...;
spawn(move || ...);
```

Since **Send** is just a normal trait, once we've written our impls, it works like any other trait. We know that this **Foo** type is **Send** because...

```
#[derive(Send)]
struct Foo<T, U>(T, U);

let foo: Foo<&Mutex<u8>, u16> = ...;
spawn(move || ...);
```

We derive **Send** for **Foo...**

```
#[derive(Send)]
struct Foo<T, U>(T, U);

let foo: Foo<&Mutex<u8>, u16> = ...;
spawn(move || ...);
```

We manually implement **Send** for **Mutex...**

```
#[derive(Send)]
struct Foo<T, U>(T, U);

let foo: Foo<&Mutex<u8>, u16> = ...;
spawn(move || ...);
```

And we manually implement it for the primitive types.

## Definition

```
unsafe trait Send
```

Thread safety property

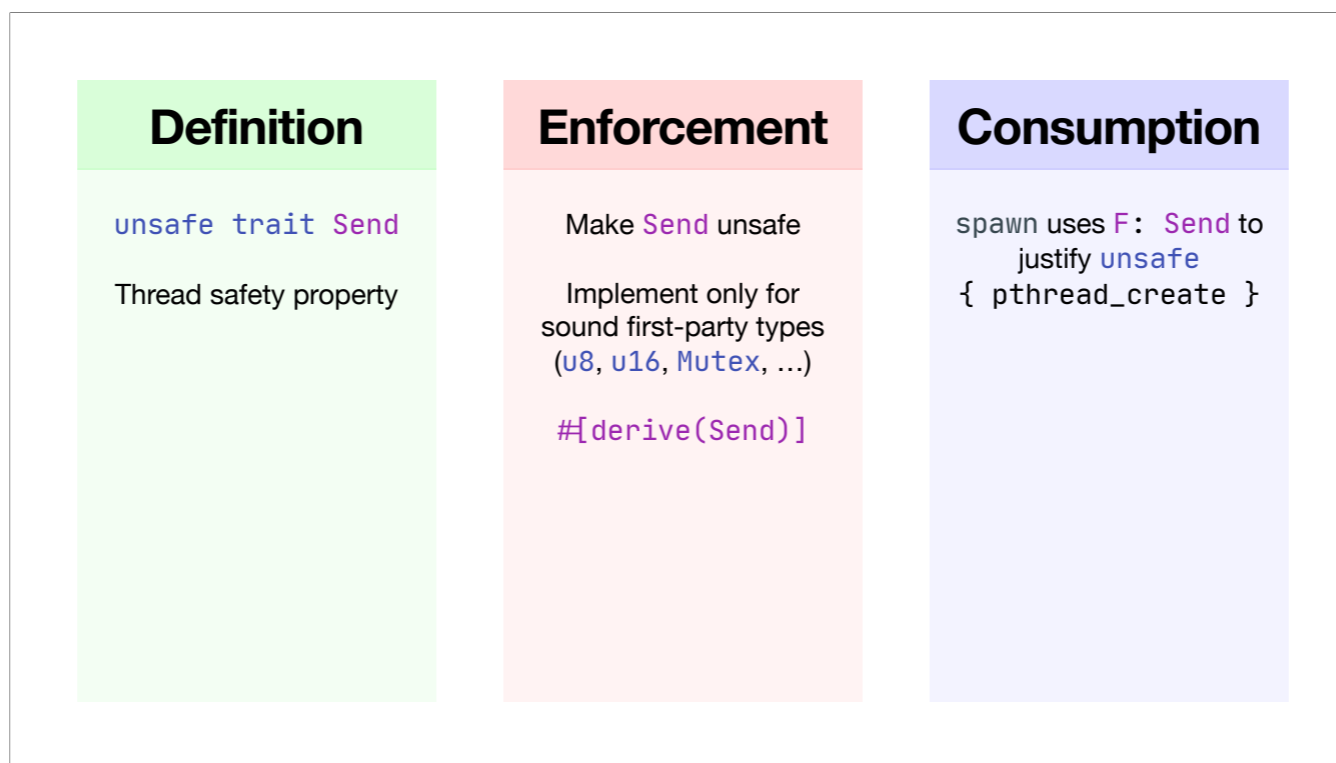
So how does this example fit into our framework?

First, we define **Send**, teaching Rust about a new trait. We use prose to document that **Send** carries a special safety property that Rust can't reason about.



Second, we enforce the safety property. We define **Send** as an **unsafe** trait so that users can't violate the safety property without writing the **unsafe** keyword. We manually implement **Send** for types which we know satisfy the safety property. And we write a custom derive that generates an implementation which guarantees thread safety.





Finally, in `spawn`, we use a `Send` bound to justify that our internal call to `pthread_create` is sound.

And that's how Rust guarantees thread safety without any support from the language itself.

# Deadlock Prevention

[github.com/akonradi](https://github.com/akonradi)

[5]

Our second example is about preventing deadlocks.

This is work by Alex Konradi, who was on the Netstack3 team at the time.

```
struct Stack {  
    ip: Mutex<IpState>,  
    device: Mutex<DeviceState>,  
}
```

Imagine you're writing a netstack, and you to make your stack multi-threaded. You care about performance, so instead of putting one mutex around all of your state, you do fine-grained locking, using small mutexes around different pieces of state.

As we saw in the previous example, Rust will guarantee that anything you do with these mutexes is thread-safe, but it won't prevent you from deadlocking.

```
struct Stack {
    ip: Mutex<IpState>,
    device: Mutex<DeviceState>,
}

// Thread A
stack.ip.lock();
stack.device.lock();

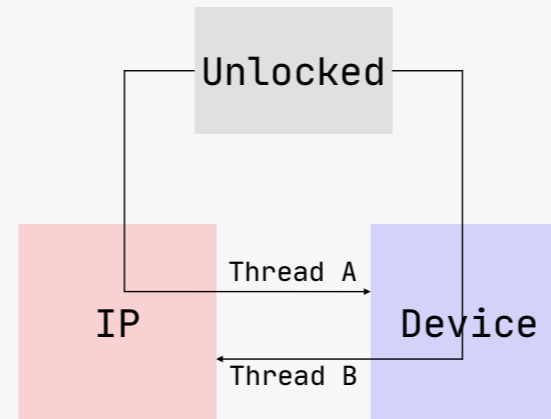
// Thread B
stack.device.lock();
stack.ip.lock();
```

For example, imagine that we have two different threads which acquire these two mutexes in different orders. Thread A acquires the IP mutex first, while thread B acquires the device mutex first.

```
struct Stack {  
    ip: Mutex<IpState>,  
    device: Mutex<DeviceState>,  
}
```

```
// Thread A  
stack.ip.lock();  
stack.device.lock();
```

```
// Thread B  
stack.device.lock();  
stack.ip.lock();
```



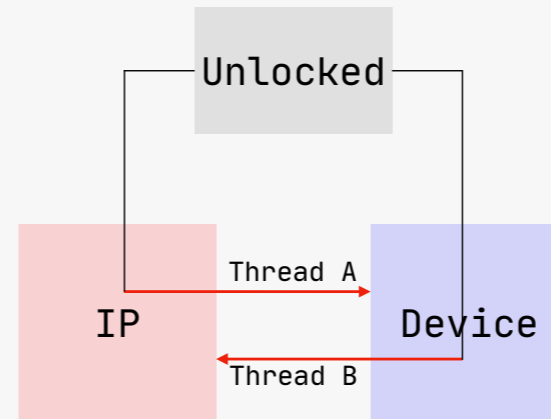
This is a classic deadlock scenario. If we're unlucky, Thread A will successfully acquire the IP mutex and Thread B will successfully acquire the device mutex. Now, in order to make progress, they're each blocked on each other.

You can conceptualize this as a graph of the locks we might want to acquire. All threads start out in the "unlocked" state, and can take different paths through the graph of locks. In order to avoid the risk of deadlocking, we need to make sure that this graph is *acyclic*.

```
struct Stack {  
    ip: Mutex<IpState>,  
    device: Mutex<DeviceState>,  
}
```

```
// Thread A  
stack.ip.lock();  
stack.device.lock();
```

```
// Thread B  
stack.device.lock();  
stack.ip.lock();
```



This set of edges forms a cycle, and so we need to remove at least one of these edges in order to make the graph acyclic.

Now this might seem like a fairly trivial problem given this example - it's pretty obvious from looking at this code that that it might deadlock.

## 77 mutexes in Netstack3

But Netstack3 has 77 different mutexes protecting all manner of different state. You can imagine that keeping track of the order in which you're supposed to acquire mutexes in that environment would get pretty difficult pretty fast.

And indeed, Netstack2, which is written in Go, has historically been plagued by deadlocks.

So the Netstack3 team decided to try to prevent deadlocks statically at compile time.

To do this, they needed a way of encoding a "lock order graph" in the type system, and a way of ensuring that code can only acquire locks consistent with that graph. Let's see how they did it.

```
struct Mutex<Id, T> {  
    mtx: std::sync::Mutex<T>,  
    _marker: PhantomData<Id>,  
}
```

The first step is to name each mutex. This new **Mutex** type is identical to the standard library mutex, except it carries an extra type parameter which is used to identify the mutex.



```
struct Stack {  
    ip: Mutex<IpState>,  
    device: Mutex<DeviceState>,  
}
```

To show how this works, let's go back to our example from before.

```
struct Stack {  
    ip: Mutex<IpLock, IpState>,  
    device: Mutex<DeviceLock, DeviceState>,  
}  
  
enum IpLock {}  
enum DeviceLock {}
```

We define one type per mutex, and use these as a sort of “name” to refer to each mutex. Note that these types are never constructed at runtime; we only need them for type system purposes.

```
/// # Safety
///
/// No cycles!
unsafe trait LockAfter<M> {}
unsafe trait LockBefore<M> {}

unsafe impl<B: LockAfter<A>, A> LockBefore<B> for A {}
```

Next, we define the **LockAfter** and **LockBefore** traits.

These are used to encode the lock ordering graph. They're unsafe to implement since the user has to promise not to introduce a lock ordering graph with cycles. If they do, they'll violate our safety property.

```

macro_rules! impl_lock_after {
  ($A:ty => $B:ty) => {
    // SAFETY: The blanket impl will cause any cycles
    // to result in a blanket impl conflict, and thus
    // won't compile.
    unsafe impl LockAfter<$A> for $B {}
    unsafe impl<X: LockBefore<$A>>
      LockAfter<X> for $B
    {
    }
  };
}

impl_lock_after!(TransportLock => IpLock);
impl_lock_after!(IpLock => DeviceLock);

```

Next, we define a macro which implements these traits on behalf of a user.

```

macro_rules! impl_lock_after {
  ($A:ty => $B:ty) => {
    // SAFETY: The blanket impl will cause any cycles
    // to result in a blanket impl conflict, and thus
    // won't compile.
    unsafe impl LockAfter<$A> for $B {}
    unsafe impl<X: LockBefore<$A>>
      LockAfter<X> for $B
    {
    }
  };
}

impl_lock_after!(TransportLock => IpLock);
impl_lock_after!(IpLock => DeviceLock);

```

Note that the macro expansion adds a blanket impl. Don't worry about following the full logic here, but the consequence of this blanket impl is important: if a user invokes the macro in such a way that introduces cycles into their lock graph, those blanket impls will conflict with one another. This means that cyclic graphs will fail to compile.

```
struct LockCtx<Id>(PhantomData<Id>);
```

Next, we define the notion of a “lock context”. A **LockCtx** allows the type system to keep track of where in the lock graph you are at any given moment. Note that it’s a zero-sized type, so it has no cost at runtime.

```
impl<Id, T> Mutex<Id, T> {
    fn lock<L>(
        &self,
        ctx: &mut LockCtx<L>
    ) → (MutexGuard<'_, T>, LockCtx<Id>)
    where
        L: LockBefore<Id>,
    {
        (
            self.mtx.lock().unwrap(),
            LockCtx(PhantomData),
        )
    }
}
```

Using these building blocks, we can finally build a deadlock-proof locking API.

Let's walk through the components of this type signature one by one.

```

impl<Id, T> Mutex<Id, T> {
    fn lock<L>(
        &self,
        ctx: &mut LockCtx<L>
    ) → (MutexGuard<'_, T>, LockCtx<Id>)
    where
        L: LockBefore<Id>,
    {
        (
            self.mtx.lock().unwrap(),
            LockCtx(PhantomData),
        )
    }
}

```

First, we require an existing **LockCtx**. Note that we borrow it mutably, which ensures that the same **LockCtx** can't be used again until the lock has been unlocked by dropping the **MutexGuard**.



```
impl<Id, T> Mutex<Id, T> {
  fn lock<L>(
    &self,
    ctx: &mut LockCtx<L>
  ) → (MutexGuard<'_, T>, LockCtx<Id>)
  where
    L: LockBefore<Id>,
  {
    (
      self.mtx.lock().unwrap(),
      LockCtx(PhantomData),
    )
  }
}
```

Second, we require that the existing **LockCtx** has a lock ID which is upstream of *our* lock ID in the graph.

```

impl<Id, T> Mutex<Id, T> {
    fn lock<L>(
        &self,
        ctx: &mut LockCtx<L>
    ) → (MutexGuard<'_, T>, LockCtx<Id>)
    where
        L: LockBefore<Id>,
    {
        (
            self.mtx.lock().unwrap(),
            LockCtx(PhantomData),
        )
    }
}

```

Finally, we return a new **LockCtx** which has our lock ID. So long as the returned **MutexGuard** exists - in other words, so long as this lock is held - this new **LockCtx** is the only **LockCtx** which is usable and it only permits locking mutexes which are downstream of this one in the lock graph.

```
enum Unlocked {}

impl LockCtx<Unlocked> {
    const UNLOCKED: LockCtx<Unlocked> = LockCtx(...);
}
```

Finally, one last bit of boilerplate: In order to represent the root of any lock graph, we introduce the **Unlocked** type, and permit anyone to construct a **LockCtx** which starts off as unlocked.

```
struct Stack {
    ip: Mutex<IpLock, IpState>,
    device: Mutex<DeviceLock, DeviceState>,
}

enum IpLock {}
enum DeviceLock {}
```

So let's go back to our example and see how we can use this new machinery to prevent deadlocks.

As we showed before, we introduce one type for each mutex.

```
struct Stack {
    ip: Mutex<IpLock, IpState>,
    device: Mutex<DeviceLock, DeviceState>,
}

enum IpLock {}
enum DeviceLock {}

impl_lock_after!(Unlocked => IpLock);
impl_lock_after!(IpLock => DeviceLock);
```

Next, we use the macro to encode our graph in the type system as implementations of the **LockAfter** and **LockBefore** traits. Remember that this code will only compile so long as the graph is free of cycles.

```
struct Stack {
    ip: Mutex<IpLock, IpState>,
    device: Mutex<DeviceLock, DeviceState>,
}

enum IpLock {}
enum DeviceLock {}

impl_lock_after!(Unlocked => IpLock);
impl_lock_after!(IpLock => DeviceLock);

// Thread A
let mut ctx = LockCtx::UNLOCKED;
```

Now let's walk through how we can use this graph at runtime. First, we construct a new context which starts off in the unlocked state.

```
struct Stack {
    ip: Mutex<IpLock, IpState>,
    device: Mutex<DeviceLock, DeviceState>,
}

enum IpLock {}
enum DeviceLock {}

impl_lock_after!(Unlocked => IpLock);
impl_lock_after!(IpLock => DeviceLock);

// Thread A
let mut ctx = LockCtx::UNLOCKED;
let (ip, mut ctx) = stack.ip.lock(&mut ctx);
```

Next, we lock our first mutex - the IP mutex. This generates a **MutexGuard** and a new **LockCtx** which shadows the old one.

```
struct Stack {
    ip: Mutex<IpLock, IpState>,
    device: Mutex<DeviceLock, DeviceState>,
}

enum IpLock {}
enum DeviceLock {}

impl_lock_after!(Unlocked => IpLock);
impl_lock_after!(IpLock => DeviceLock);

// Thread A
let mut ctx = LockCtx::UNLOCKED;
let (ip, mut ctx) = stack.ip.lock(&mut ctx);
let (device, mut ctx) = stack.device.lock(&mut ctx);
```

Finally, we repeat the same process to lock the device mutex.



```
struct Stack {
    ip: Mutex<IpLock, IpState>,
    device: Mutex<DeviceLock, DeviceState>,
}

enum IpLock {}
enum DeviceLock {}

impl_lock_after!(Unlocked => IpLock);
impl_lock_after!(IpLock => DeviceLock);

// Thread A
let mut ctx = LockCtx::UNLOCKED;
let (ip, mut ctx) = stack.ip.lock(&mut ctx);
let (device, mut ctx) = stack.device.lock(&mut ctx);

// Thread B
let mut ctx = LockCtx::UNLOCKED;
```

Now let's take a look at what happens if we try to violate the lock order. As before, we start off in an unlocked state.

```
struct Stack {
    ip: Mutex<IpLock, IpState>,
    device: Mutex<DeviceLock, DeviceState>,
}

enum IpLock {}
enum DeviceLock {}

impl_lock_after!(Unlocked => IpLock);
impl_lock_after!(IpLock => DeviceLock);

// Thread A
let mut ctx = LockCtx::UNLOCKED;
let (ip, mut ctx) = stack.ip.lock(&mut ctx);
let (device, mut ctx) = stack.device.lock(&mut ctx);

// Thread B
let mut ctx = LockCtx::UNLOCKED;
let (device, mut ctx) = stack.device.lock(&mut ctx);
```

But now, we directly lock the device mutex. Now, on its own, this is fine since we haven't introduced a cycle yet, and so this compiles.

```
struct Stack {
    ip: Mutex<IpLock, IpState>,
    device: Mutex<DeviceLock, DeviceState>,
}

enum IpLock {}
enum DeviceLock {}

impl_lock_after!(Unlocked => IpLock);
impl_lock_after!(IpLock => DeviceLock);

// Thread A
let mut ctx = LockCtx::UNLOCKED;
let (ip, mut ctx) = stack.ip.lock(&mut ctx);
let (device, mut ctx) = stack.device.lock(&mut ctx);

// Thread B
let mut ctx = LockCtx::UNLOCKED;
let (device, mut ctx) = stack.device.lock(&mut ctx);
let (ip, mut ctx) = stack.ip.lock(&mut ctx); // ← ERROR
```

But if, having acquired the device mutex, we now try to acquire the IP mutex, that won't compile since there's no path in the graph from the device mutex to the IP mutex.

## Definition

```
unsafe trait  
  LockAfter
```

```
unsafe trait  
  LockBefore
```

```
LockAfter, LockBefore  
define acyclic order
```

```
struct Mutex<Id, T>
```

```
Mutex only locked  
consistent with defined  
order
```

So let's fit this into our framework.

First, we define the **LockAfter** and **LockBefore** traits. We document in prose that they carry a safety property - that they always encode an acyclic graph.

Second, we define the **Mutex** type. We document in prose that it carries a safety property - that it is only locked consistent with a pre-defined lock order graph.

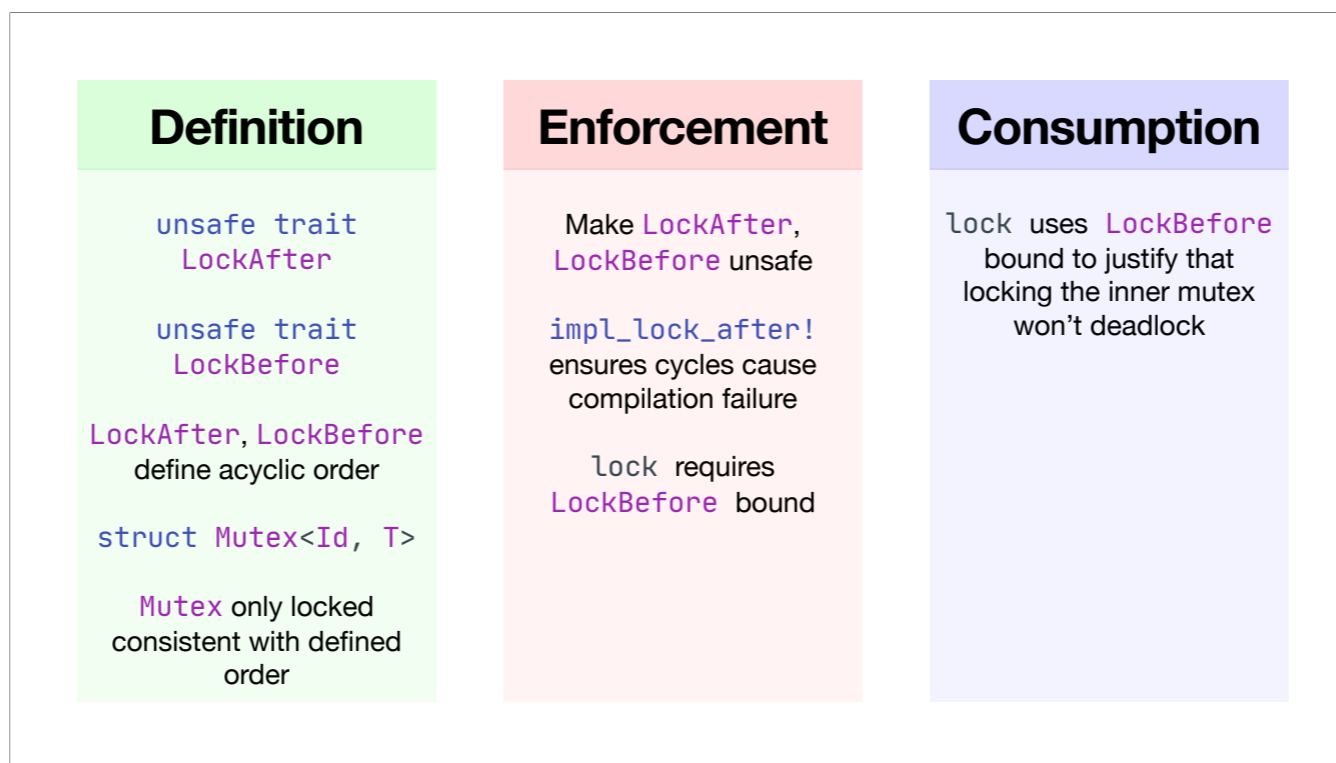
Definition	Enforcement
<pre> unsafe trait   LockAfter  unsafe trait   LockBefore  LockAfter, LockBefore define acyclic order  struct Mutex&lt;Id, T&gt;  Mutex only locked consistent with defined order </pre>	<pre> Make LockAfter, LockBefore unsafe  impl_lock_after! ensures cycles cause compilation failure  lock requires LockBefore bound </pre>

Next, enforcement.

We make the **LockAfter** and **LockBefore** traits unsafe so that a user can only violate our safety property by writing the **unsafe** keyword.

We design the **impl\_lock\_after** macro so that it guarantees that only acyclic graphs can compile.

We use a **LockBefore** bound to ensure that the **lock** method can only be called consistent with the lock order graph.



Finally, consumption.

The **lock** method uses its **LockBefore** bound to justify that locking the inner mutex won't deadlock.

—

Now I should note that what I've presented here is a simplified version of what exists in practice for the purposes of this presentation. There are actually two subtle ways that I'm aware of that you can violate deadlock safety using this simplified version.

There's an interesting conversation to be had about whether safety properties like these should be literally impossible to circumvent, or whether they just need to be hard enough to circumvent that you just couldn't shoot yourself in the foot by accident. In reality, the lock ordering library takes the latter approach. But I'm not going to get into that discussion here, and it doesn't really affect the point that I'm trying to make with this talk.

# The Results

So how has this played out for Netstack3? Well, in the beginning, Netstack3 was entirely single threaded for simplicity. But over the past few years, they've been working to thread mutexes and mutex guards through the entire stack. Finally, just a few weeks ago, they had finished that process and were ready to flip the switch - to go from running on one thread to running on multiple threads.

[netstack3] Default to 4 threads

- Self(NonZeroU8::new(1).unwrap())  
+ Self(NonZeroU8::new(4).unwrap())

fxrev.dev/1102436

[6]

So they changed the default number of threads from one to four. That's it. That was the whole change. It was bug-free on the first try.

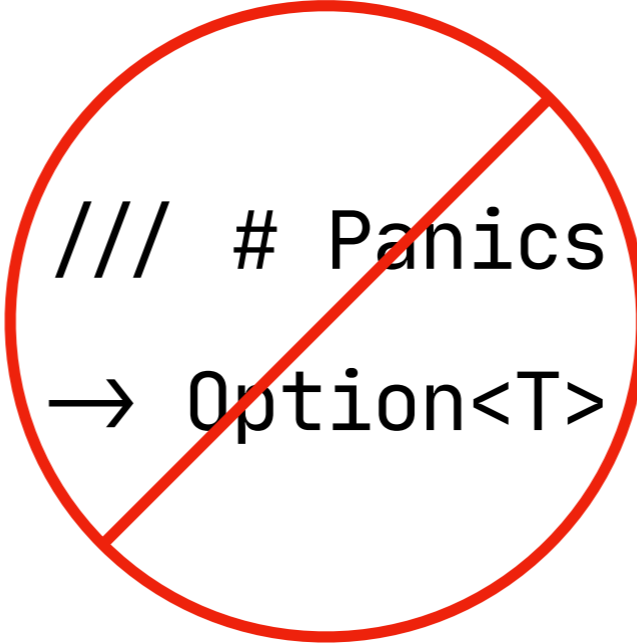


# Conclusions

So those are my two examples.

I'd like to encourage you all to try this methodology for yourself, and especially to bring it to new domains and new classes of bugs that nobody in the Rust community has tackled yet.

Let me give you a few pieces of concrete advice.



```
/// # Panics  
→ Option<T>
```

On a practical note, think of partial functions in your API as a code smell. Your public functions should never panic, and they usually shouldn't return **Option** or **Result** unless that's inherent to the behavior you're implementing. Returning an error because an IO operation failed is fine, but returning an error because the caller passed an illegal value is not. That code shouldn't have compiled in the first place.

Parse, don't validate

*Alexis King*

Type Safety Back and Forth

*Matt Parsons*

[7]

For great primers, I'd recommend these blog posts.

```

pub enum IpParseError<I: IcmpIpExt> {
  Parse { error: ParseError },
  ParameterProblem {
    src_ip: I::Addr,
    dst_ip: I::Addr,
    code: I::ParameterProblemCode,
    pointer: I::ParameterProblemPointer,
    must_send_icmp: bool,
    header_len: I::HeaderLen,
    action: IpParseErrorAction,
  },
}

pub enum IpParseErrorAction {
  DiscardPacket,
  DiscardPacketSendIcmp,
  DiscardPacketSendIcmpNoMulticast,
}

pub enum ParseError {
  NotSupported,
  NotExpected,
  Checksum,
  Format,
}

```

Make your APIs exactly match the structure of the problem you're modeling.

Here we see the error type that is returned when Netstack3 fails to parse an IP packet.

This may seem surprisingly convoluted, but this is a faithful representation of the complexity of **both** the IPv4 and IPv6 protocols. To do anything simpler would cause us problems down the line.

```
pub enum IpParseError<I: IcmpIpExt> {
  Parse { error: ParseError },
  ParameterProblem {
    src_ip: I::Addr,
    dst_ip: I::Addr,
    code: I::ParameterProblemCode,
    pointer: I::ParameterProblemPointer,
    must_send_icmp: bool,
    header_len: I::HeaderLen,
    action: IpParseErrorAction,
  },
}
```

For example, let's zoom in on the "parameter problem pointer".

If you receive a malformed packet, you have to respond with your own packet which contains an error message. That error message identifies the byte offset of the malformed field that caused parsing to fail. That byte offset is known as the "parameter problem pointer."

Note the generic type here. In IPv4, the field in the error message that stores this pointer is one byte long, while in IPv6, it's four bytes long. Using generics here ensures we always store the byte offset using the right size of integer. If we didn't do this, we'd have to do a fallible conversion elsewhere in the codebase, and if we had a bug, that conversion would either silently produce incorrect error packets, or it would panic and crash the stack.

By ensuring that our error type faithfully models the behavior of IPv4 and IPv6, we can ensure that this sort of bug is impossible.

## Diversity

Also don't expect the solution to always be the same. Just because the catch phrase is always that "buggy programs don't compile" doesn't mean that there's a recipe. In my experience, different problems call for wildly different solutions that leverage a hodgepodge of different language features.

**“Simplicity is the art of hiding complexity.”**

**Rob Pike**

[8]

And don't expect it to always fall out naturally. One of my favorite quotes is from Rob Pike talking about Go's design: "Simplicity is the art of hiding complexity."

Your job is to do hard work so that your users can have an easy mental model of your API. Rust wasn't explicitly designed to support this methodology, at least not in its full generality, and you may have to really bend the language to your will.

For example, if you haven't seen the trick, it's not at all obvious that it would be possible to prevent graph cycles using the trait system. The *internals* of the lock ordering library are kind of ugly as a result, but the mental model that a user needs in order to reason about the API is very simple: cyclic graphs don't compile. That's it.

## Turtles all the way down

Next, when it comes to your *internal* APIs, use the same level of discipline and rigor that you would use with a *public* API. If a function has safety requirements, make it an unsafe function. If a function can panic, document it clearly. It may seem obvious in the moment how to call an internal API correctly, but it won't be so obvious to that new developer on the team four years and six refactors from now. If you keep up this discipline, then no matter how much time passes, your internal APIs will be just as easy to use correctly as the day you wrote them.

We adhered to this discipline strictly in Netstack3, and the larger the project grew, the more it paid off.



**“We're late in the podcast now, so we can let our hair down a bit and speculate.”**

**Sean Carroll**

And finally, if you'll permit me a bit of speculation, I think this methodology has the potential to fundamentally reshape how we engage in the process of software engineering, and I think that so far we've only scratched the surface with how far we can push it.

One reason is that there are domains where correctness is a much bigger deal than it is for most of us in this room. Rust is only just starting to make inroads into high-assurance domains like automotive, aerospace, medical devices, and so on. Those are domains where bugs are so unacceptable that the pace of development is absolutely glacial. I'm not suggesting we're going to be able to just stop doing code reviews or anything crazy like that, but imagine how much faster we could move if we eliminated entire classes of bugs that we normally have to catch through reviews or testing?

Another reason that I think this methodology could be a big deal is that it allows software to scale more effectively. Software complexity often scales super-linearly as a function of the size of the codebase, and much of that complexity is incidental rather than inherent. Humans only have so much mental capacity, so as the incidental complexity increases, it crowds out our ability to reason about the inherent complexity that we're actually interested in. That puts a natural upper limit on the inherent complexity of the problems that we can tackle.

My guess is that this methodology stands to make the biggest difference in large, complex codebases where small savings in complexity across every module compound on one another. That's certainly been our experience in Netstack3.

[tinyurl.com/safety-rustconf-24](https://tinyurl.com/safety-rustconf-24)

References



As I mentioned at the beginning, I wasn't able to provide nearly as many examples as I'd have liked to. I really encourage you to check out the references and do more reading on your own. Obviously I'm also happy to talk in person or on Discord.

I really believe that, thanks to Rust, we have the opportunity to fundamentally improve the art of software engineering, and I intend to spend the next phase of my career proving it. I hope you'll join me. Thank you very much.