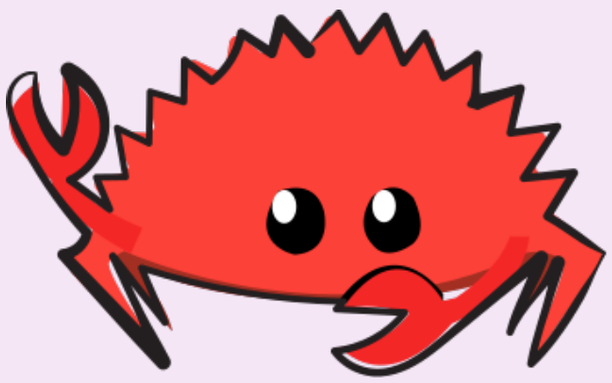


Safety Goggles *for* Alchemists

Jack Wrenn | RustConf 2024

Thank you for the introduction!

About Me

- Applied Scientist at AWS
- Co-Maintainer of Itertools
- Rider of Tandem Bicycles
- Very Friendly!
- Lead of Project Safe Transmute
- Also Have a History Degree

jack.wrenn.fyi
jack@wrenn.fyi

To tell you a bit more about myself:
- I'm currently an Applied Scientist at AWS
- I co-maintain itertools and a few other crates you might've heard of.
- I am an avid tandem bike tourer.
- I'm very friendly — come say "Hi!" in the hallway!

And, relevant to this talk,
- I lead Project Safe Transmute
- And was a history major…

So let's talk history and transmute!

History of Rust

As some of you may know, when Grayson Hoare first presented Rust at the Mozilla Annual Summit in 2010, his opening slide described the project as “*Technology from the past, come to save the future from itself.*”

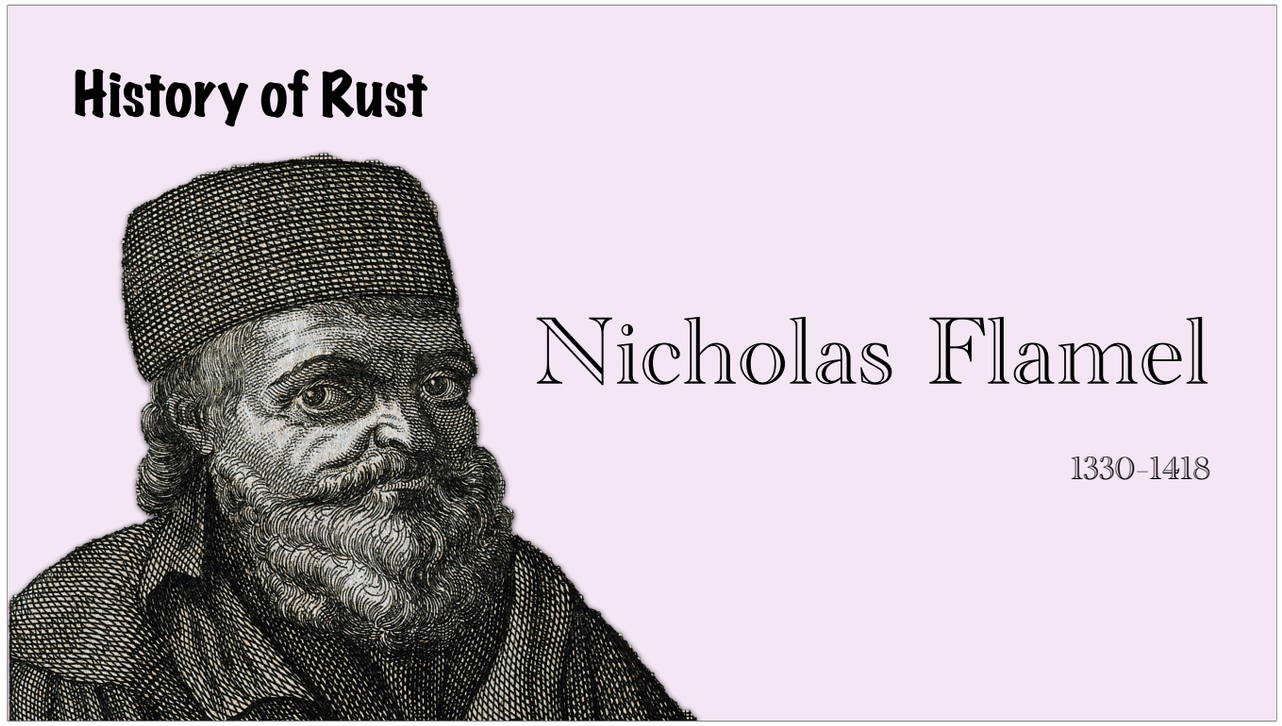
History of Rust

Rust's Influences

- **SML**, **OCaml**: algebraic data types, pattern matching, type inference, semicolon statement separation
- **C++**: references, RAII, smart pointers, move semantics, monomorphization, memory model
- **ML Kit**, **Cyclone**: region based memory management
- **Haskell (GHC)**: typeclasses, type families
- **Newsqueak**, **Alef**, **Limbo**: channels, concurrency
- **Erlang**: message passing, thread failure, linked thread failure, lightweight concurrency
- **Swift**: optional bindings
- **Scheme**: hygienic macros
- **C#**: attributes
- **Ruby**: closure syntax, block syntax
- **NIL**, Hermes: typestate
- **Unicode Annex #31**: identifier and pattern syntax

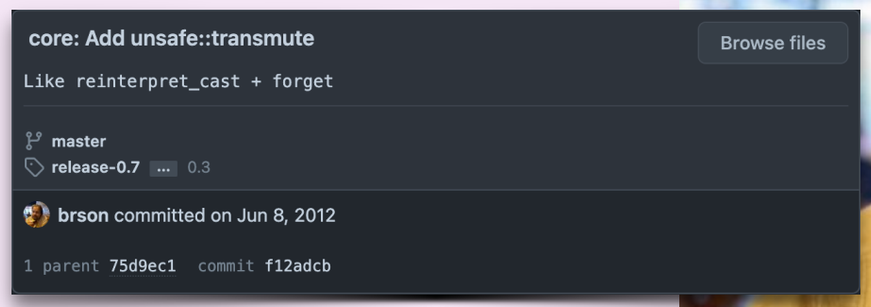
This is usually interpreted as a nod to the past sixty years of programming language development that Rust draws from; work like StandardML, Scheme, and, famously, Cyclone.

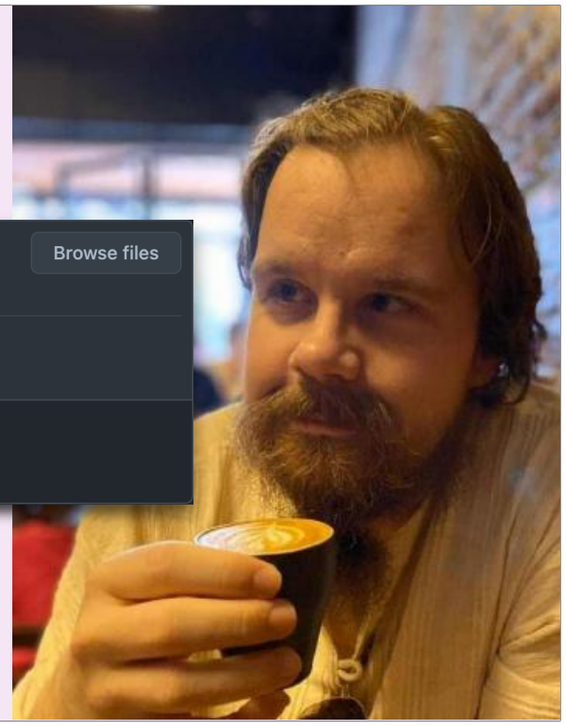
But to *really* understand Rust's roots, we actually need to turn back the clock a little further…



…to 1379! That year — so the story goes — famed bookseller Nicholas Flamel finally translated a cryptic tome he had purchased twenty two years earlier: the Book of Abramelin the Mage. With its occult insights, he transmuted lead into gold and even — some say — cracked the code of immortality. Perhaps took his secrets to his grave, or perhaps he assumed a false identity. All I know is that six hundred years later…

History of Rust





…another bearded man began contributing to an upstart programming language called Rust. This is “Bryan Anderson”, and in July 2012, he added a curious new function called `transmute`.

Alchemy in Rust

```
unsafe fn transmute<Src, Dst>(src: Src) -> Dst
{
    ...
}
```

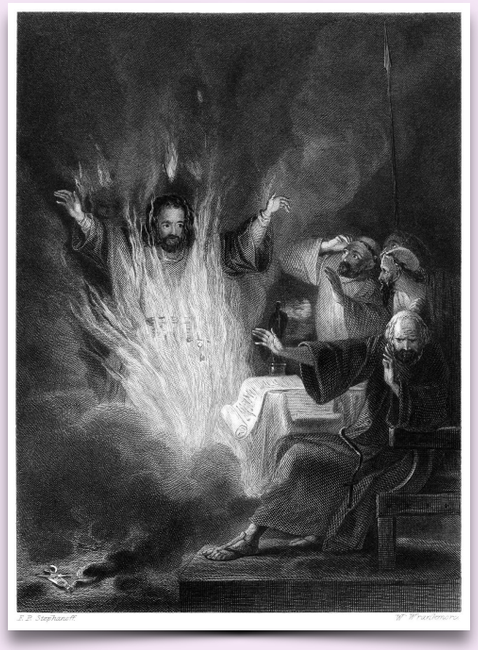

Transmute is an unsafe function that takes a source value of any type of your choosing, and returns a destination type of your choosing backed by those same bits.

Alchemy in Rust

```
unsafe fn transmute<Src, Dst>(src: Src) -> Dst
{
    ...
}

struct Lead;
struct Gold;

let gold: Gold = unsafe { transmute(Lead) };
```


Transmute is a universal constructor, and with it, Rust can transmute lead into gold, and, perhaps, even gain immortality.

Risks



…but with great power, comes even greater danger. Let’s enumerate four major risks. There are others, but these are the four need to worry about unless your dealing with function pointers or DSTs.

1. Bit Validity

```rust
fn u8_to_bool(src: u8) -> bool {
    // UNSOUND!
    unsafe { transmute::<u8, Bool>(src) }
}
```


Foremost, you must guarantee that the source value is a valid instance of the destination type. Consider this function function, which transmutes a `u8` to a `bool`.

The layout of a `u8` is a single, initialized byte of any value. The layout of a `bool` is a single, initialized byte with the value of either zero or one.

This function, therefore, is sound to invoke only with `0` or `1`. What happens if you pass in a `2`? Undefined behavior.

2. Alignment

```
fn u8s_to_u16(src: &[u8; 2]) -> &u16 {
    // UNSOUND!
    unsafe { transmute::<&[u8; 2], &u16>(src) }
}
```


Next, if references are involved, you need to worry about at least two things in addition to their bit validity. The first is reference alignment.

Types in Rust have alignment requirements. Whereas `u8`s can appear at any memory address, `u16`s can only appear at *even* memory addresses on most platforms.

Invoking this function, which transmutes a reference to two `u8`s into a reference to a `u16`, will be sound about half the time. It's 50/50 whether the source slice has an even memory address.

Getting this wrong is undefined behavior.

3. Lifetime Extension

```
fn extend<'a>(src: &'a u8) -> &'static u8 {
    // UNSOUND!
    unsafe { transmute::<&'a _, &'static _>(src) }
}
```


And of course, wherever there are references, there are lifetimes. It's trivial to use transmute to extend a reference with a bounded lifetime into a reference with a static lifetime. Although there *are* valid use-cases for this, it's easy to accidentally end up with a dangling reference and that, of course, is undefined behavior.

4. Safety Invariants

Declaration

```rust
pub struct Even {
    // SAFETY: Always an even number!
    n: u8
}

impl Even {
    fn new(n: u8) -> Option<Self> {
        …
    }
}
```

Use

```rust
fn u8_to_even(src: u8) -> Even {
    // UNSOUND!
    unsafe { transmute::<u8, Even>(src) }
}
```

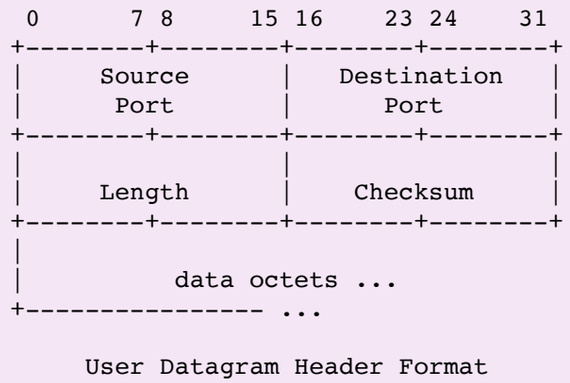

And finally, most subtle of all, you need to worry about violating your own safety invariants. Let’s say one person on your team defines a struct that holds even numbers and enforces that with a checked constructor, and then you bypass that constructor with `transmute`.

Unless you’ve guaranteed that your source value upholds all the safety invariants of the destination type, this is potentially unsound!

Rewards

So given these risks, why do people bother with `transmute` at all?

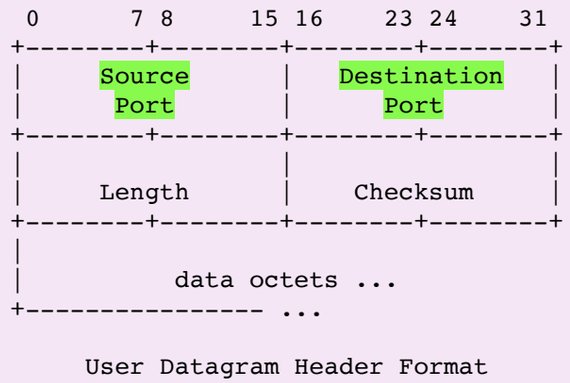
Case Study: Packet Parsing



User Datagram Header Format

As a case study, let’s consider a simplified scenario of parsing UDP packets. A UDP header looks something like this. Two bytes for the source port, another two for the destination port, another two for the length, two for the checksum, and then the data follows.

Case Study: Packet Parsing



User Datagram Header Format

To keep our examples fitting on the screen, we'll look at just parsing out the first two fields: the source and destination ports.

Case Study: Packet Parsing

```
struct UdpPacketHeader {
    src_port: u16,
    dst_port: u16,
}

fn read_header(bytes: &[u8]) -> Option<UdpPacketHeader> {
    let Some((&src_port, bytes)) = bytes.split_first_chunk() else {
        return None
    };

    let Some((&dst_port, bytes)) = bytes.split_first_chunk() else {
        return None
    };

    Some(UdpPacketHeader {
        src_port: u16::from_be_bytes(src_port),
        dst_port: u16::from_be_bytes(dst_port),
    })
}
```


A traditional parser might look something like this. The `read_header` function on screen consumes an input slice of bytes, and returns an optional, owned `UdpPacketHeader`.

Case Study: Packet Parsing

```
struct UdpPacketHeader {
    src_port: u16,
    dst_port: u16,
}

fn read_header(bytes: &[u8]) -> Option<UdpPacketHeader> {
    let Some((&src_port, bytes)) = bytes.split_first_chunk() else {
        return None
    };

    let Some((&dst_port, bytes)) = bytes.split_first_chunk() else {
        return None
    };

    Some(UdpPacketHeader {
        src_port: u16::from_be_bytes(src_port),
        dst_port: u16::from_be_bytes(dst_port),
    })
}
```


It works by first reading out the first two bytes of the input by value — that's the source port.

Case Study: Packet Parsing

```
struct UdpPacketHeader {
    src_port: u16,
    dst_port: u16,
}

fn read_header(bytes: &[u8]) -> Option<UdpPacketHeader> {
    let Some((&src_port, bytes)) = bytes.split_first_chunk() else {
        return None
    };

    let Some((&dst_port, bytes)) = bytes.split_first_chunk() else {
        return None
    };

    Some(UdpPacketHeader {
        src_port: u16::from_be_bytes(src_port),
        dst_port: u16::from_be_bytes(dst_port),
    })
}
```


Then by reading out the second two bytes by value — that's the destination port.

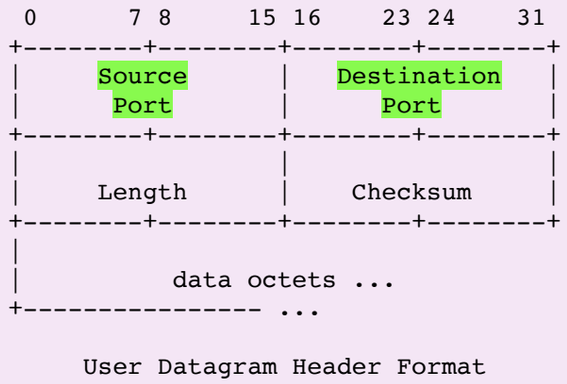
Case Study: Packet Parsing

```rust
struct UdpPacketHeader {
    src_port: u16,
    dst_port: u16,
}

fn read_header(bytes: &[u8]) -> Option<UdpPacketHeader> {
    let Some((&src_port, bytes)) = bytes.split_first_chunk() else {
        return None
    };

    let Some((&dst_port, bytes)) = bytes.split_first_chunk() else {
        return None
    };

    Some(UdpPacketHeader {
        src_port: u16::from_be_bytes(src_port),
        dst_port: u16::from_be_bytes(dst_port),
    })
}
```


…and finally it packages up this data, by value, into an owned `UdpPacketHeader`.

This is all well and good, but with `transmute`, we can do a lot better.

Case Study: Packet Parsing




```rust
#[repr(C)]
struct UdpPacketHeader {
    src_port: [u8; 2],
    dst_port: [u8; 2],
}
```


The specification for UDP is describing the wire layout of a packet, and with very little effort, we can define a Rust datatype that has the exact same layout.

Case Study: Packet Parsing

```
#[repr(C)]
struct UdpPacketHeader {
    src_port: [u8; 2],
    dst_port: [u8; 2],
}

fn view_header(bytes: &[u8]) -> Option<&UdpPacketHeader> {
    let Ok(bytes) = <&[u8; 4]>::try_from(bytes) else {
        return None;
    };

    Some(unsafe { transmute(bytes) })
}
```


Given this, we don't need to copy any bytes out of our network buffer. We can, instead, define `view_header`, which takes our reference-to-bytes and turns it into a reference-to-a-`UdpPacketHeader`.

Case Study: Packet Parsing

```rust
#[repr(C)]
struct UdpPacketHeader {
    src_port: [u8; 2],
    dst_port: [u8; 2],
}

fn view_header(bytes: &[u8]) -> Option<&UdpPacketHeader> {
    let Ok(bytes) = <&[u8; 4]>::try_from(bytes) else {
        return None;
    };

    Some(unsafe { transmute(bytes) })
}
```


It does so by first checking that there enough bytes in our buffer,

Case Study: Packet Parsing

```rust
#[repr(C)]
struct UdpPacketHeader {
    src_port: [u8; 2],
    dst_port: [u8; 2],
}

fn view_header(bytes: &[u8]) -> Option<&UdpPacketHeader> {
    let Ok(bytes) = <&[u8; 4]>::try_from(bytes) else {
        return None;
    };

    Some(unsafe { transmute(bytes) })
}
```


And then, if so, transmutes that reference-to-bytes to a reference-to-`UdpPacketHeader`.

This is called “zero copy parsing”, and not only did it require far fewer lines of code than `read_header`…

Case Study: Packet Parsing

`read_header` (copying)

```
read_header:
    cmpq $4, %rsi
    jae .LBB0_3
    xorl %eax, %eax
    jmp .LBB0_2

.LBB0_3:
    movzwl (%rdi), %ecx
    rolw $8, %cx
    movzwl 2(%rdi), %edx
    rolw $8, %dx
    movw $1, %ax

.LBB0_2:
    movzwl %dx, %edx
    shlq $32, %rdx
    shll $16, %ecx
    orq %rdx, %rcx
    movzwl %ax, %eax
    orq %rcx, %rax
    retq
```

`view_header` (zero-copy)

```
view_header:
        xor     eax, eax
        cmp     rsi, 4
        cmove   rax, rdi
        ret
```


…it optimizes down to a quarter of the amount machine code. Scaled across an entire networking stack, this approach can provide tremendous performance advantages.

And remember, for the purpose of demonstration, we ignored parsing out all of the other fields of UDP packet header. If we parsed those out too, the complexity of read_header would, accordingly, grow linearly. The complexity of view_header, however, remains constant.

Case Study: Packet Parsing

```
#[repr(C)]
struct UdpPacketHeader {
    src_port: [u8; 2],
    dst_port: [u8; 2],
}

fn view_header(bytes: &[u8]) -> Option<&UdpPacketHeader> {
    let Ok(bytes) = <&[u8; 4]>::try_from(bytes) else {
        return None;
    };

    Some(unsafe { transmute(bytes) })
}
```

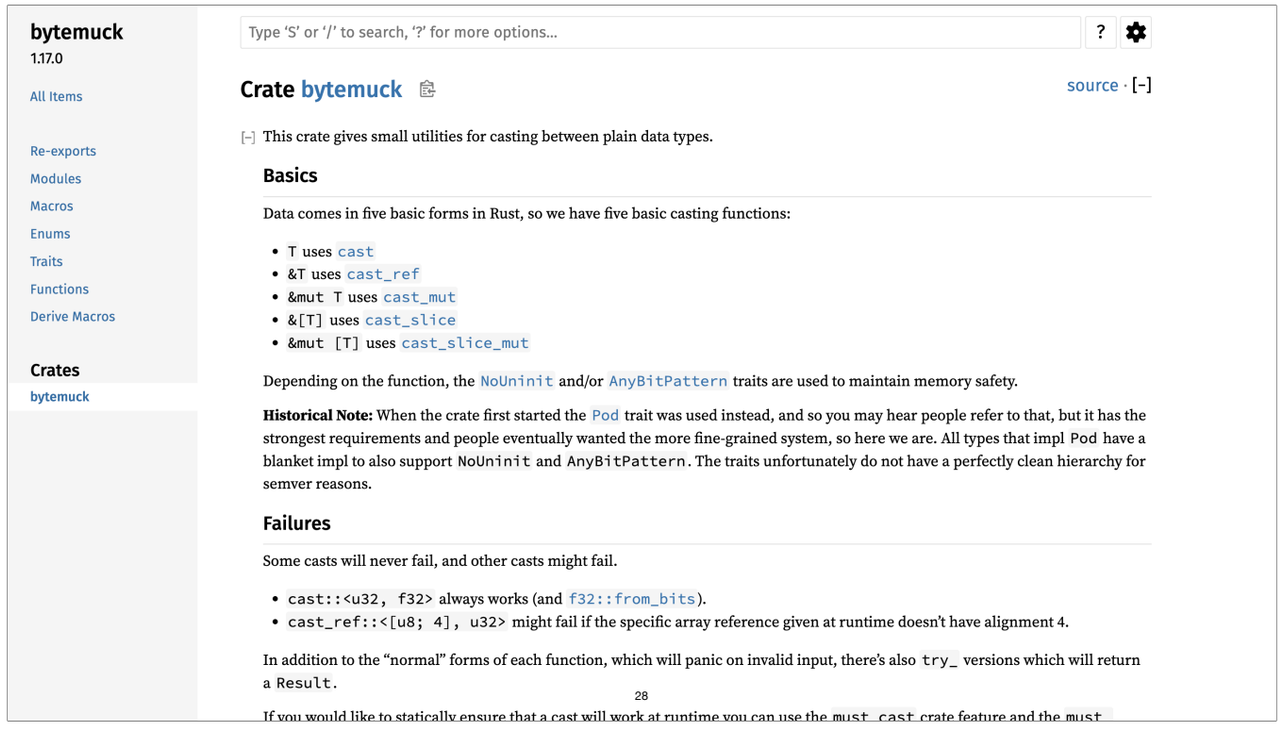

…but it's also tremendously dangerous. What if we forgot our `repr(C)` annotation? What if we used a field type with a greater alignment requirement?

The penalty of getting any details wrong is undefined behavior, and trusting individual programmers to get them right isn't scalable.

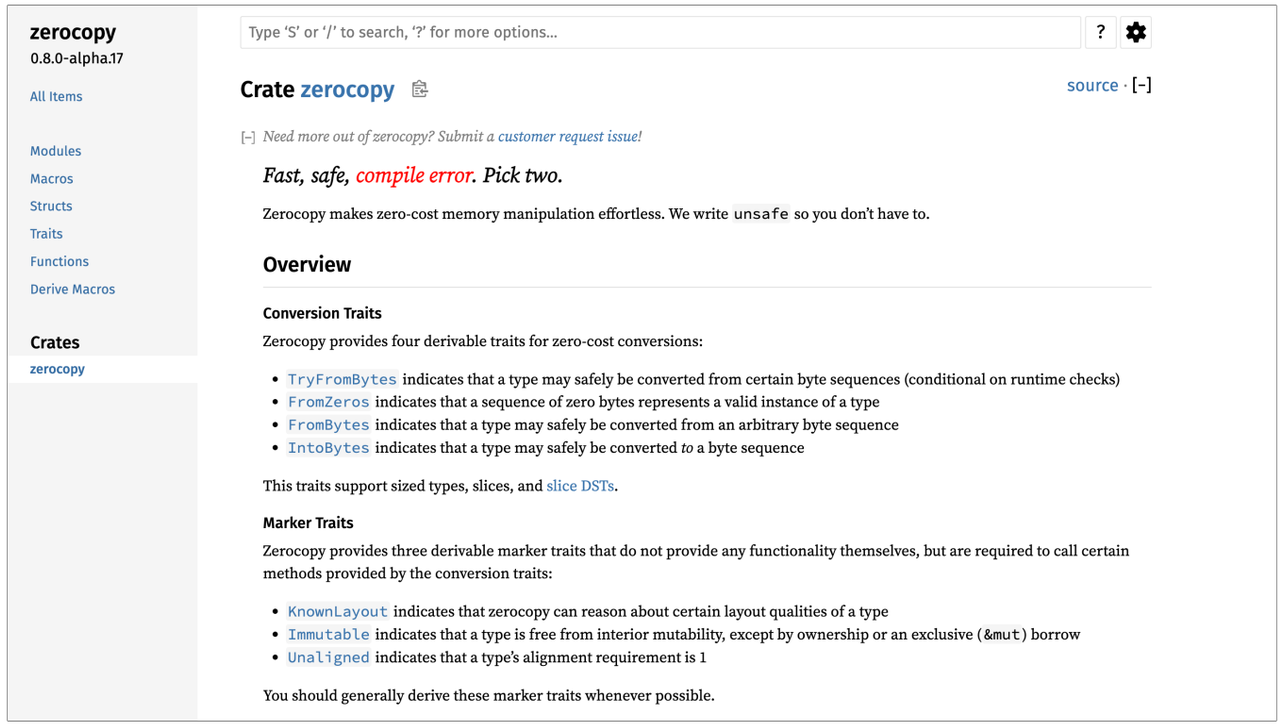
Transmutation Crates

The response to this in the ecosystem has been the creation of various crates that provide safe abstractions for transmutation. They don’t prevent all mistakes, but they do prevent the ones that lead to memory unsafely.

The two major, exceptional, players in this area are



Bytemuck, whose foundational abstraction is “plain old data” that can be freely converted to and from raw bytes, plus a slew of other goodies.



…and Zerocopy, which provides a hierarchy of four conversion marker traits, all implementing various kinds of safe transmute to and from bytes. I'm going to pick on zerocopy a lot in this talk, because I help maintain it.

Inside Safe Transmutation Crates

Marker Traits and Inductive Reasoning

```rust
/// Types for which any bit pattern is valid.
///
/// # Safety
///
/// `Self` must be soundly transmutable from
/// initialized bytes.
unsafe trait FromBytes { … }
```

unsafe marker trait

```rust
unsafe impl FromBytes for f32 {}
unsafe impl FromBytes for f64 {}
unsafe impl FromBytes for i8 {}
unsafe impl FromBytes for i16 {}
unsafe impl FromBytes for i32 {}
unsafe impl FromBytes for i64 {}
unsafe impl FromBytes for i128 {}
unsafe impl FromBytes for isize {}
/* and so on */
```

base implementations

Both of these crates address safe transmutation with marker traits. These marker traits are unsafe, denoting that they carry invariants which you must prove to be true when implementing them, and which you can rely on while writing unsafe code.

And for each trait, these crates typically provide a comprehensive set of base implementations over primitive types.

For example, zerocopy's `FromBytes` marker trait denotes types which can be safely initialized from arbitrary bytes. All of the primitive numbers are `FromBytes`, but types like `bool` and `char` are not, and so implementations for them are intentionally omitted.

Inside Safe Transmutation Crates

Marker Traits and Inductive Reasoning

```rust
/// Types for which any bit pattern is
valid.
///
/// # Safety
///
/// `Self` must be soundly transmutable from
/// initialized bytes.
unsafe trait FromBytes { … }
```

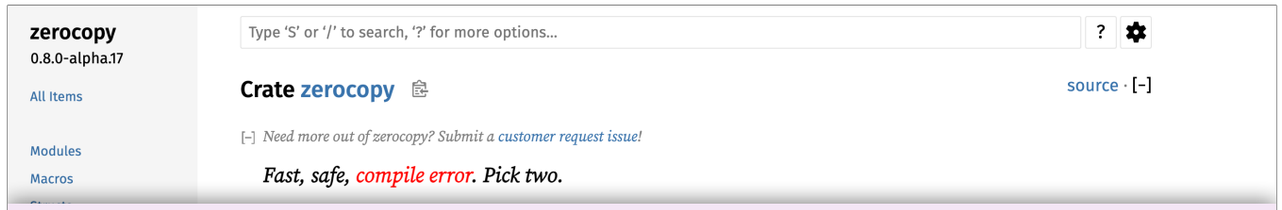
unsafe marker trait

```rust
unsafe impl FromBytes for (A, B)
where
    A: FromBytes,
    B: FromBytes,
{}
/* and so on */
```

inductive, compound implementations

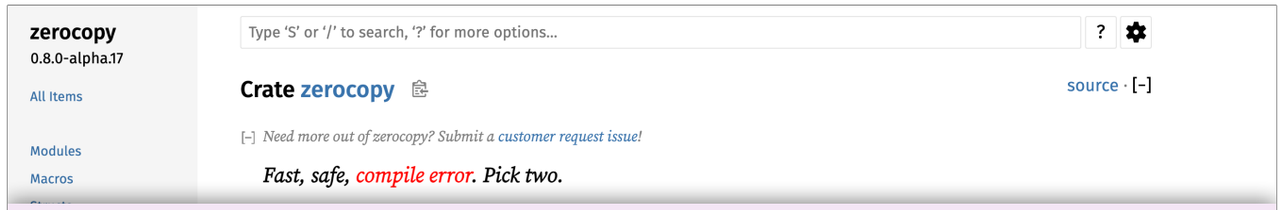
To support compound types, these crates combine inductive reasoning and additional checks. For example, a struct or tuple is `FromBytes` if its fields are all `FromBytes`.

Fortunately, as a user, you don’t need to remember any of these rules or write unsafe code yourself. Both bytemuck and zerecopy provide proc macro derives for their marker traits.



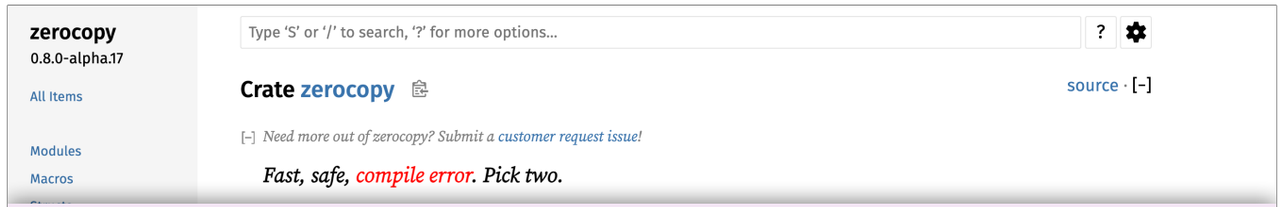

```rust
use zerocopy::{FromBytes, Immutable, KnownLayout};


#[repr(C)]
struct UdpPacketHeader {
    src_port: [u8; 2],
    dst_port: [u8; 2],
}

fn view_udp_packet(bytes: &[u8]) -> Option<&UdpPacketHeader> {
   …
}
```


So, if we were to apply zerocopy to our packet parsing problem, we wouldn't actually manually implement these traits for `UdpPacketHeader`.




```rust
use zerocopy::{FromBytes, Immutable, KnownLayout};

#[derive(FromBytes, Immutable, KnownLayout)]
#[repr(C)]
struct UdpPacketHeader {
    src_port: [u8; 2],
    dst_port: [u8; 2],
}

fn view_udp_packet(bytes: &[u8]) -> Option<&UdpPacketHeader> {
   …
}
```

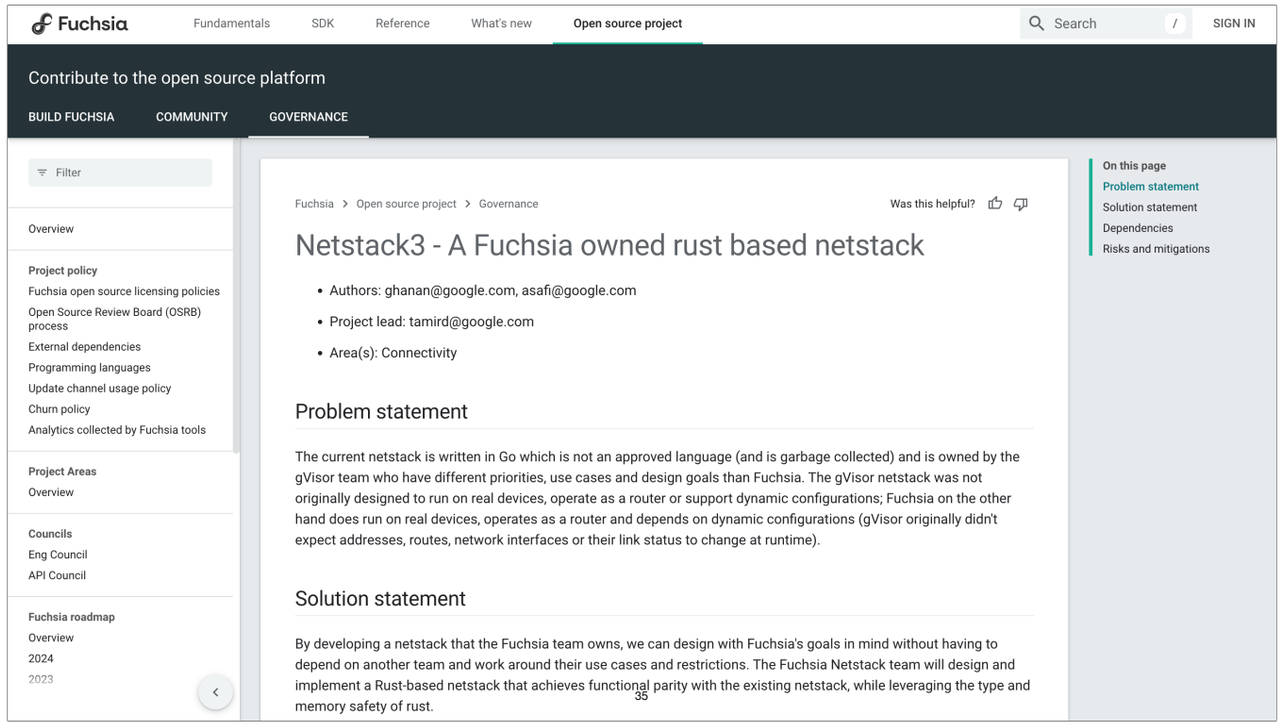

Rather, we just derive the right traits…




```rust
use zerocopy::{FromBytes, Immutable, KnownLayout};

#[derive(FromBytes, Immutable, KnownLayout)]
#[repr(C)]
struct UdpPacketHeader {
    src_port: [u8; 2],
    dst_port: [u8; 2],
}

fn view_udp_packet(bytes: &[u8]) -> Option<&UdpPacketHeader> {
    UdpPacketHeader::ref_from_bytes(bytes).ok()
}
```


…and call the safe methods that they provide. Now, we're down to just a single line of code.

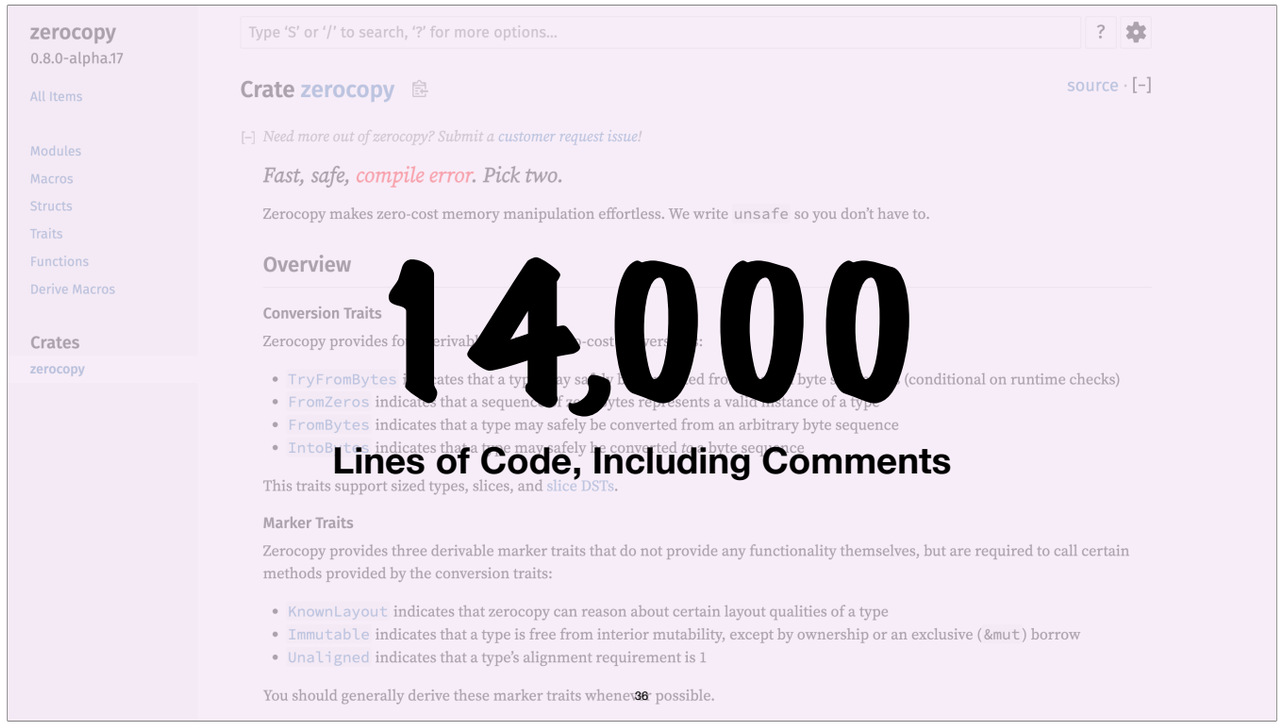
With these abstractions, you can build incredible things.



At Google, engineers working on their new Fuchsia operating system, used the zero copy crate to build a networking stack that parses and serializes packets almost entirely in-place and achieved this feat almost bug-free.

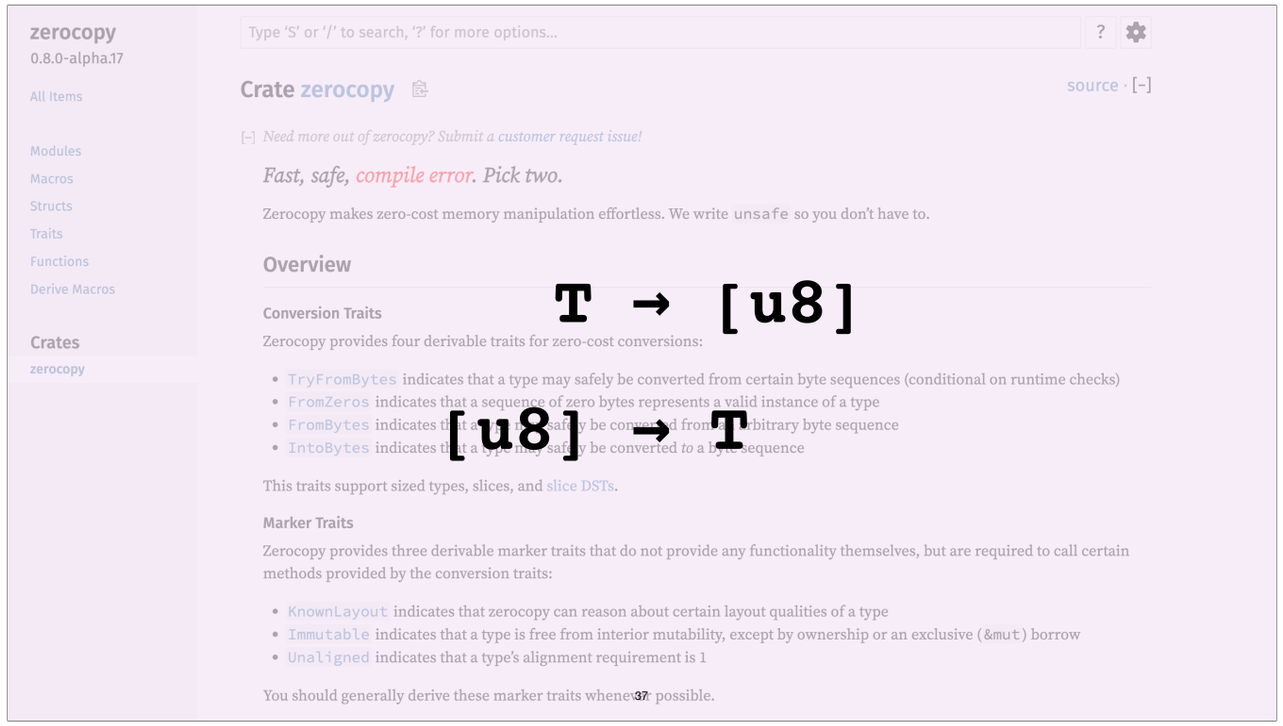
If you’re curious about how they accomplished this feat, I’d encourage you to attend Josh Liebow-Feeser’s talk tomorrow on Safety in an Unsafe World.

Suffice to say, the zero copy crate provided the foundational abstractions that made this possible. The previous slide wasn’t just a cutesy talk example — it’s more-or-less what the netstack code looks like!



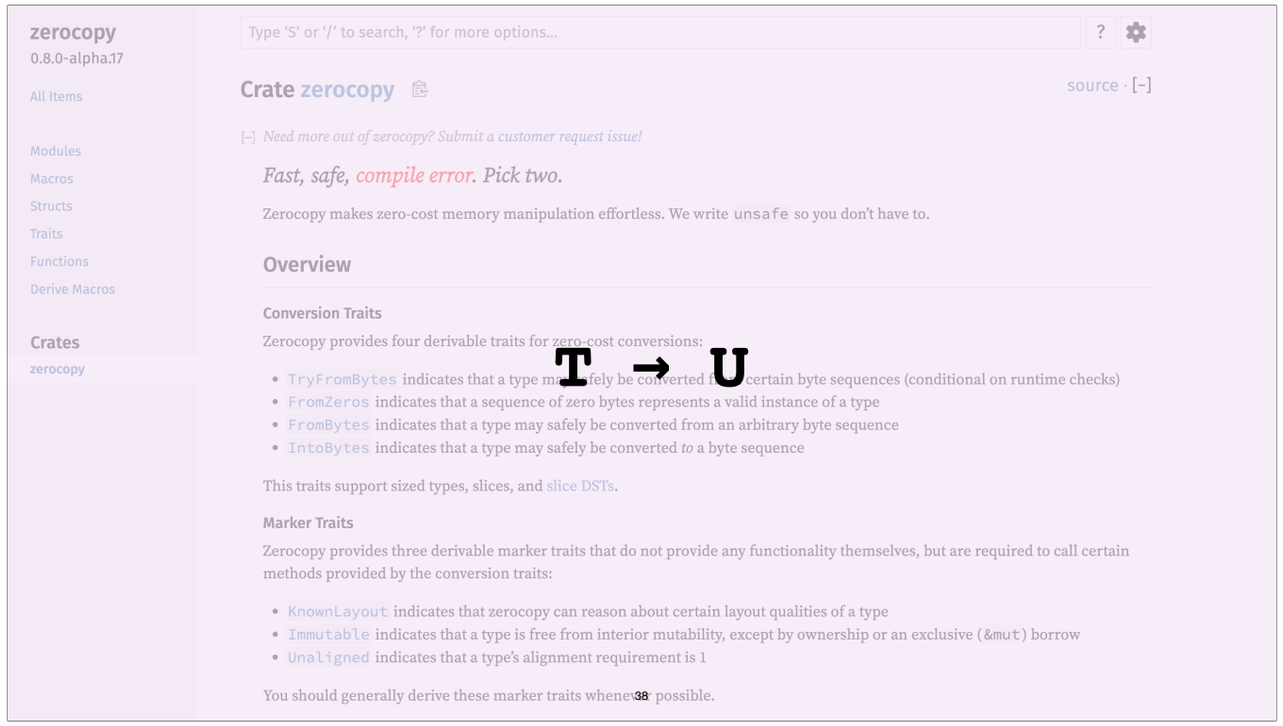
But zerocopy's dirty secret is that it's backed by nearly fourteen *thousand* lines of subtle unsafe code and comments.

This is, insane. The compiler *already* knows about the layouts of types, so why is the onus entirely on you to reason about them in the context of transmute? Why can't the compiler use its knowledge to tell you when one type is transmutable into another?



And remember, zerocopy only solves the limited problems of safe transmutes into bytes, and from bytes.

If state-of-the-art networking stacks can be built by solving this limited problem…



…what could be enabled by an analysis that handles transmutations between arbitrary types?

Project Safe Transmute

RFC 2835

These are the questions that motivated the creation of Project Safe Transmute, which seeks to extend the Rust compiler with these reasoning abilities.

Theory of Alchemy

To begin to do so, we had to develop a theory of type alchemy. The general idea is quite straight-forward:

Theory of Alchemy

A type, **`Src`***, is transmutable into a type,* **`Dst`***, if every possible value of* **`Src`** *is a valid value of* **`Dst`***.*

"*A type,* `Src`*, is transmutable into a type,* `Dst`*, if every possible value of Src, is a valid value of Dst.*"

At first glance, this sounds like a simple problem of sets and subsets.

Theory of Alchemy

Observation: Type are Sets of Values

```
       layout(u8) = {0x00, ... 0xFF}
layout(NonZeroU8) = {0x01, ... 0xFF}


layout(NonZeroU8) ⊆ layout(u8)
layout(NonZeroU8) ⊉ layout(u8)
```


For example, we can think of the layout of a `u8` as being the set of all initialized bytes ranging from `0` to `255`.

And the layout of a `NonZeroU8` as being all values ranging from `1` to `255`.

Given this, we can say that `NonZeroU8` *is* transmutable into `u8` because its layout is a subset of that of `u8`, and that the reverse is not transmutable, because `u8` is a superset of `NonZeroU8`.

Theory of Alchemy

Observation: Sets are Computationally Expensive

$$|\texttt{layout(u8)}| = 2^8 = 256$$
$$|\texttt{layout(u16)}| = 2^{16} = 65{,}536$$
$$|\texttt{layout(u32)}| = 2^{32} = 4{,}294{,}967{,}296$$
$$|\texttt{layout(u64)}| = 2^{64} = 18{,}446{,}744{,}073{,}709{,}551{,}616$$

But this representation is computationally expensive. If a type is a single byte, its layout is a set of at most 256 elements. But as the size of the type grows, the maximum size of its layout set grows exponentially.

Theory of Alchemy

Observation: Types May Not Be Finite

```
struct LinkedList<T> {
    head: T,
    tail: Option<Box<T>>,
}
```

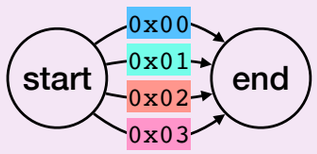
$$|\texttt{layout(LinkedList<u8>)}| = \infty$$

Worse, types are not always finite. A linked list doesn’t have a bounded size. It might even be circular!

Theory of Alchemy

Observation: Types are Automata

```
#[repr(u8)]
enum Direction {
    North,
    East,
    South,
    West
}
```



Fortunately, there’s another way we can view types.

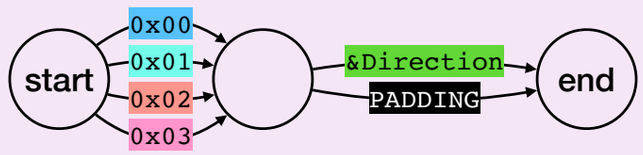
The layout of a type is a finite automaton, in which each edge represents aa validity constraint, and each path of edges from start to finish represents a particular possible value of the type.

Here, the direction enum has four possible bit valid instances, and, accordingly, there are four paths through its corresponding automaton.

Theory of Alchemy

Observation: Automata Are Expressive

```
#[repr(C)]
struct LinkedList {
    head: Direction,
    tail: Option<Box<Direction>>,
}

#[repr(usize)]
enum Direction {
    North,
    East,
    South,
    West
}
```



This representation is extremely expressive. In addition to bit values, edges can encode padding, or references. The automaton for a linked list of directions looks something like.

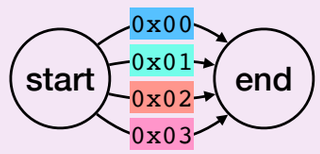
I’m glossing over some details about interfield padding here, but I hope you get the gist.

And best yet, this representation scales gracefully.

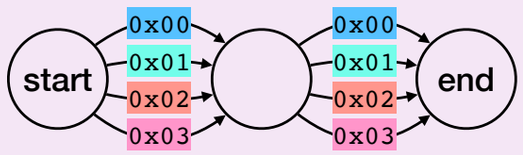
Theory of Alchemy

Observation: Automata Scale Gracefully

```
#[repr(u8)]
enum Direction {
    North,
    East,
    South,
    West
}
```




```
[Direction; 2]
```



16 possible values

only 8 edges

For example, the number of edges in an array of `Direction`s scales linearly with length, even though the total number of paths grows exponentially.

At first glance, this might seem like it doesn't buy us much.

Theory of Alchemy

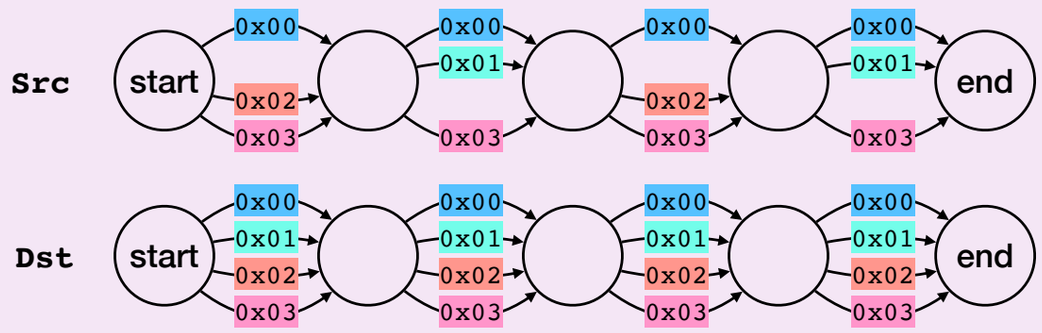
Observation: Transmutation with Automata is Simple

*A type, **`Src`**, is transmutable into a type, **`Dst`**, if every possible path in **`Src`** is a path in **`Dst`**.*

If we, reframe our theory of alchemy in terms of paths the problem still seems quite hard: "*A `Src` is transmutable into a `Dst` if every possible path in source is also a path in `Dst`.*"

Theory of Alchemy

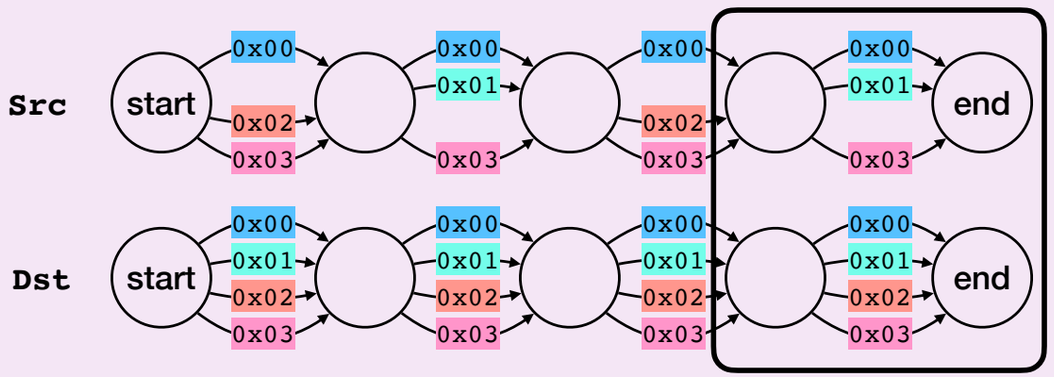
Observation: Transmutation with Automata is Efficient



But thanks to the structure of the graph representation, we can solve transmutability in polynomial time.

Theory of Alchemy

Observation: Transmutation with Automata is Efficient

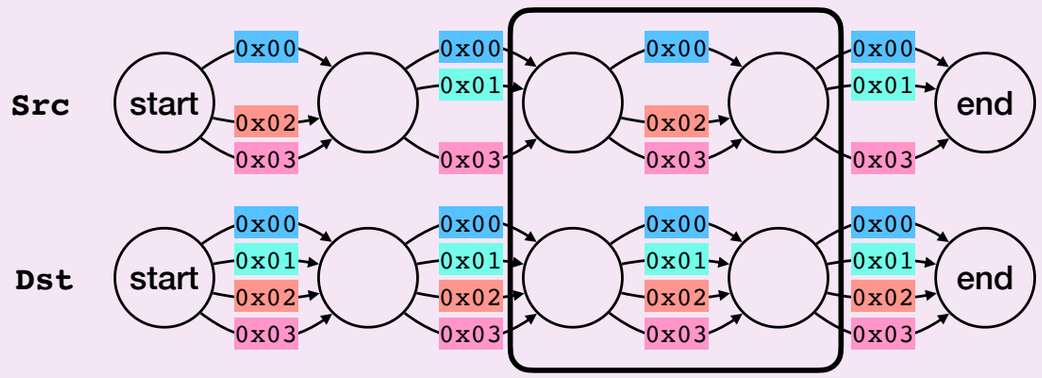


*this is a sketch

Applying dynamic programming, we simply start from byte-offsets of from the end of the type..

Theory of Alchemy

Observation: Transmutation with Automata is Efficient

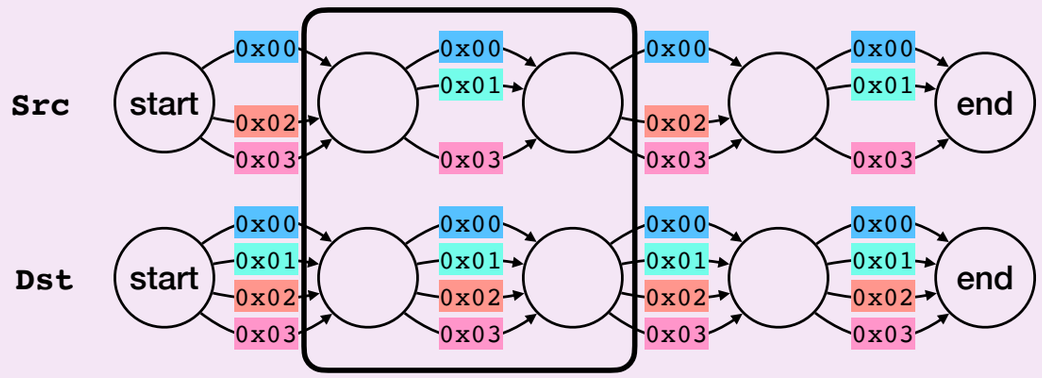


*this is a sketch

…and march…

Theory of Alchemy

Observation: Transmutation with Automata is Efficient

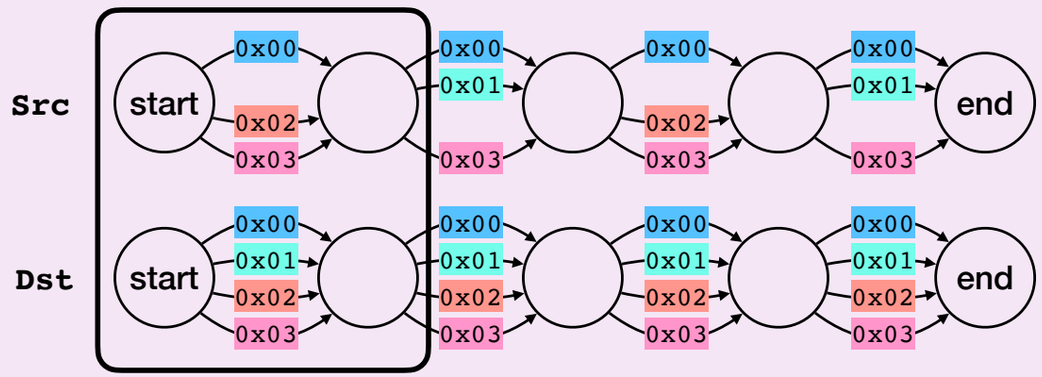


*this is a sketch

…our way…

Theory of Alchemy

Observation: Transmutation with Automata is Efficient



*this is a sketch

…to the start, answering the transmutability question incrementally.

Thanks to this insight, solving transmutability for arbitrary types is both theoretically possible and practical to implement.

Dst: TransmuteFrom<Src>

MCP411 • #[feature(transmutability)]

```rust
unsafe trait TransmuteFrom<Src: ?Sized> {
    unsafe fn transmute(src: Src) -> Self
    where
        Src: Sized,
        Self: Sized;
}
```


So we implemented it, in a trait called `TransmuteFrom`, which is automatically implemented by the compiler, on-the-fly, for any two types that are transmutable.

It takes in a `Src` type, and, for convenience, it has an associated `transmute` method.

Safe Alchemy

From Comments to Compile Errors

With this trait, our examples of unsoundness become compilation errors. Let’s take a look.

1. Bit Validity

Program

```
fn u8_to_bool(src: u8) -> bool {
    // compile error!
    unsafe { TransmuteFrom::transmute(src) }
}
```

Compiler Output

```
error[E0277]: `u8` cannot be safely transmuted into `bool`
 --> src/lib.rs:3:38
  |
4 |     unsafe { TransmuteFrom::transmute(src) }
  |              ------------------------ ^^^ at least one value of
  |              |                          `u8` isn't a bit-valid
  |              |                          value of `bool`
  |              required by a bound introduced by this call
```


If we rewrite our bit validity example to use this trait and method, it fails to compile. And the compilation error explains exactly why: `u8` has at least one value that isn't a bit-valid instance of `bool`.

2. Alignment

Program

```
fn u8s_to_u16(src: &[u8; 2]) -> &u16 {
    // compile error!
    unsafe { TransmuteFrom::transmute(src) }
}
```

Compiler Output

```
error[E0277]: `&[u8; 2]` cannot be safely transmuted into `&u16`
 --> src/lib.rs:3:38
  |
4 |     unsafe { TransmuteFrom::transmute(src) }
  |              ------------------------ ^^^ the minimum alignment of
  |              |                        `&[u8; 2]` (1) should be
  |              |                        greater than that of
  |              |                        `&u16` (2)
  |              required by a bound introduced by this call
```


Likewise, if we attempt a reference transmutation that might have alignment issues, we'll get an error message telling us exactly that. Here, it lets us know that the minimum alignment of a `u8` array is less than that of a `u16`.

3. Lifetime Extension

Program

```
fn extend<'a>(src: &u8) -> &'static u8 {
    // compile error!
    unsafe { TransmuteFrom::transmute(src) } }
}
```

Compiler Output

```
error: lifetime may not live long enough
 --> src/lib.rs:9:14
  |
1 | fn extend<'a>(src: &'a u8) -> &'static u8 {
  |           -- lifetime `'a` defined here
2 |     // compile error!
3 |     unsafe { TransmuteFrom::transmute(src) }
  |              ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ returning this value
  |                                             requires that `'a`
  |                                             must outlive `'static`
```


And if we try to extend the lifetime of a reference, that, too is a compilation error.

4. Safety Invariants

Upstream

```rust
pub struct Even {
    // SAFETY: An even number!
    n: u8
}
```

Downstream

```rust
fn u8_to_even(src: u8) -> Even {
    // compile error!
    unsafe { TransmuteFrom::transmute(src) }
}
```

Compiler Output

```
error[E0277]: `u8` cannot be safely transmuted into `Even`
 --> src/lib.rs:3:38
  |
4 |     unsafe { TransmuteFrom::transmute(src) }
  |              ------------------------ ^^^ `Even` may carry
  |              |                          safety invariants
  |              |
  |              required by a bound introduced by this call
```


And finally, this trait prevents us from accidentally violating the library safety invariants of types, telling us here that our `Even` type may carry safety invariants that aren't satisfied.

Now, you might be wondering: Rust doesn't yet have a concept of `unsafe` fields. So how does the compiler know that `Even` has safety invariants?

It doesn't. Any user-defined type might have safety invariants, so the only sound thing to do is assume they *all* have safety invariants.

But isn't this a little restrictive? Doesn't this mean you can't use safe transmute on user-defined types? And if the talk ended here, the answer to both of those questions would be yes.

Safer Alchemy

It turns out, “Safe” transmutation isn’t all that useful. What you really need, for most real-word applications, is *safer* transmutation.

TransmuteFrom & Assume

```rust
unsafe trait TransmuteFrom<Src: ?Sized, const ASSUME: Assume> {
    unsafe fn transmute(src: Src) -> Dst
    where
        Src: Sized,
        Self: Sized;
}
```

```rust
struct Assume {
    alignment: bool,
    lifetimes: bool,
    safety:    bool,
    validity:  bool,
}
```

`Assume` lets you configure compile-time safety checks

So we built in an escape hatch. We added a const generic parameter to `TransmuteFrom` called `Assume`, which lets you tell the compiler which safety properties it should assume that you, the programmer, are taking care of ensuring.

In essence, it lets you relax the compile-time checks of `TransmuteFrom`, so long as you promise to do those checks yourself.

`Assume::VALIDITY`

Before

```
fn u8_to_bool(src: u8) -> bool {
    // compile error!
    unsafe { TransmuteFrom::transmute(src) }
}
```

After

```
fn u8_to_bool(src: u8) -> bool {
    assert!(src < 2);
    // SAFETY: We have checked that `src` is a bit-valid
    // instance of `bool`.
    unsafe {
        TransmuteFrom::<_, Assume::VALIDITY>::transmute(src)
    }
}
```


For example, let's again consider transmuting a `u8` to a `bool`. Sure, this transmutation is invalid for most values of `u8`, but not for *all* values of `u8`. If we tell the compiler to assume validity, it will accept this transmutation instead of emitting a compilation error.

`Assume::ALIGNMENT`

Before

```
fn u8s_to_u16(src: &[u8; 2]) -> &u16 {
    // compile error!
    unsafe { TransmuteFrom::transmute(src) }
}
```

After

```
fn u8s_to_u16(src: &[u8; 2]) -> &u16 {
    assert!(<*const _>::is_aligned_to(src, align_of::<u16>()));
    // SAFETY: We've asserted that `src` meets the alignment
    // requirements of `u16`
    unsafe {
        TransmuteFrom::<_, Assume::ALIGNMENT>::transmute(src)
    }
}
```


Likewise, if we tell the compiler to assume that we've ensured alignment, it will let us transmute between references with potentially incompatible alignment requirements, again, assuming that we've done the work of guaranteeing that alignment is actually satisfied — in this case with a runtime check.

`Assume::SAFETY`

Before

```
fn u8_to_even(src: u8) -> Even {
    // compile error!
    unsafe { TransmuteFrom::transmute(src) }
}
```

After

```
fn u8_to_even(src: u8) -> Even {
    assert_eq!(src % 2, 0);
    // SAFETY: We have checked that `src` is a safe
    // instance of `Even`.
    unsafe {
        TransmuteFrom::<_, Assume::SAFETY>::transmute(src)
    }
}
```


And finally, if tell `TransmuteFrom` to assume safety, it will let us do transmutations that could violate the safety invariants of the involved types, enabling TransmuteFrom to accept transmutations involving user-defined types.

Safety Goggles for Alchemists

`#[feature(transmutability)]`

```rust
unsafe trait TransmuteFrom<Src: ?Sized, const ASSUME: Assume> {
    unsafe fn transmute(src: Src) -> Dst
    where
        Src: Sized,
        Self: Sized;
}
```

AVAILABLE NOW!

```rust
struct Assume {
    alignment: bool,
    lifetimes: bool,
    safety:    bool,
    validity:  bool,
}
```


This trait, which is available for testing now on nightly, provides effective safety goggles for folks doing transmutations and, especially, for abstracting over transmutation.

Safety Goggles for Alchemists

`#[feature(transmutability)]`

```rust
unsafe trait TransmuteFrom<Src: ?Sized, const ASSUME: Assume> {
    unsafe fn transmute(src: Src) -> Dst
    where
        Src: Sized,
        Self: Sized;
}
```

```rust
struct Assume {
    alignment: bool,
    lifetimes: bool,
    safety:    bool,
    validity:  bool,
}
```

AVAILABLE NOW!

`Assume` tells you what you need to prove in your `SAFETY` comments for `transmute`

For one-off transmutations, the value of `ASSUME` tells you exactly what you need to write in your `SAFETY` comment.

Safety Goggles for Transmute

```
// SAFETY: We have checked that:
// 1. `src` validly aligned for `Dst`
// 2. `src` is a bit-valid instance of `Dst`
unsafe {
    TransmuteFrom::<_, {
        Assume::ALIGNMENT.and(Assume::VALIDITY)
    }>::transmute(src)
}
```


For example, if we assume alignment and validity, that’s our cue that our safety comment needs to justify why both these safety conditions are satisfied.

Safety Goggles for Abstraction

```rust
/// # Safety
///
/// By implementing this, you promise that `Self`
/// can be transmuted from arbitrary init bytes.
pub unsafe trait FromBytes
{}
```

```rust
/// # Safety
///
/// By implementing this, you promise that
/// `Self` has no uninitialized bytes.
pub unsafe trait IntoBytes
{}
```


But, moreover, we see this truly versatile foundation for abstractions over transmute.

For example, implementing the `FromBytes` trait from zero copy requires proving that `Self` can be soundly transmuted from initialized bytes. With `TransmuteFrom`, this burden can be offloaded entirely onto the compiler,

Safety Goggles for Abstraction

```rust
/// # Safety
///
/// By implementing this, you promise that
/// `Self` has no safety invariants.
pub unsafe trait FromBytes: Sized
where
    Self: TransmuteFrom<[u8; size_of::<Self>()], { Assume::SAFETY }>
{}

/// # Safety
///
/// By implementing this, you promise that
/// `Self` has no uninitialized bytes.
pub unsafe trait IntoBytes
{}
```


just by adding a `where` clause, instructing the compiler to ensure that `Self` really is transmutable from a buffer of `u8`s. All the programmer has to do now, is promise that they don't have any safety invariants on the type.

With this, we think can entirely get rid of zerocopy's proc-macro derives.

Safety Goggles for Abstraction

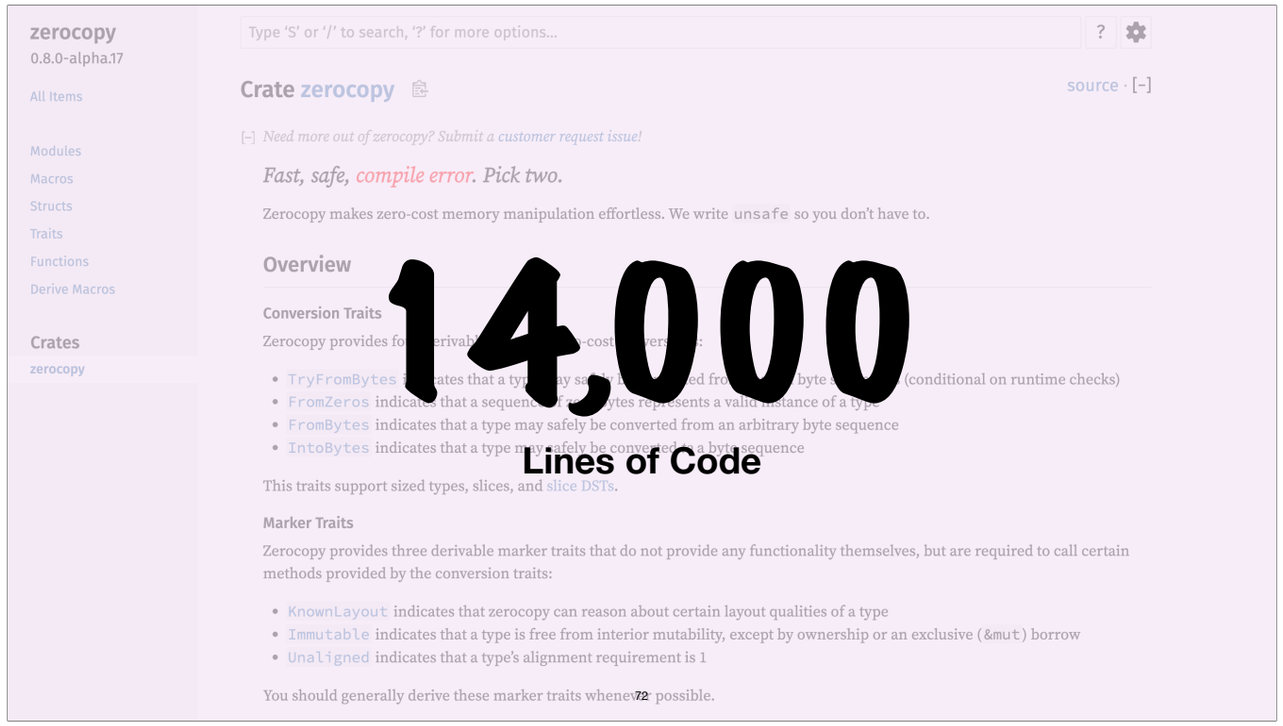
```
/// # Safety
///
/// By implementing this, you promise that
/// `Self` has no safety invariants.
pub unsafe trait FromBytes: Sized
where
    Self: TransmuteFrom<[u8; size_of::<Self>()], { Assume::SAFETY }>
{}

/// # Safety
///
/// By implementing this, you promise that
/// `Self` has no uninitialized bytes.
pub unsafe trait IntoBytes
{}
```


For `IntoBytes`, the change is even more drastic.

Safety Goggles for Zerocopy

```rust
/// # Safety
///
/// By implementing this, you promise that
/// `Self` has no safety invariants.
pub unsafe trait FromBytes: Sized
where
    Self: TransmuteFrom<[u8; size_of::<Self>()], { Assume::SAFETY }>
{}

/// # Safety
///
/// By implementing this, you promise that
/// `Self` has no uninitialized bytes.
pub trait IntoBytes: Sized
where
    [u8; size_of::<Self>()]: TransmuteFrom<Self>
{}
```


With an added `where` clause that encodes `Self` must be transmutable into initialized bytes, this trait becomes entirely safe to implement — the compiler can fully check its safety invariant.



Given this, it's with little exaggeration that think we can get zerocopy's 14,000 lines of code…

…down to about 1,400.

Future Outlook

We need your help!

But there are three remaining pieces of design and development work needed to get safe transmute over the finish line.

Support for DSTs

```
/// # Safety
///
/// By implementing this, you promise that
/// `Self` has no safety invariants.
pub unsafe trait FromBytes: Sized
where
    Self: TransmuteFrom<[u8; size_of::<Self>()], { Assume::SAFETY }>
{}

pub trait IntoBytes: Sized
where
    [u8; size_of::<Self>()]: TransmuteFrom<Self>
{}
```


Foremost, to fully support crates like Zerocopy, we need to support dynamically sized types. This demo I showed a few slides back required adding `Sized` bounds to zero copy's marker traits. Zerocopy doesn't have `Sized` bounds, and it doesn't want to add them.

What we want to to support, instead, is something like this:

Support for DSTs

```
/// # Safety
///
/// By implementing this, you promise that
/// `Self` has no safety invariants.
pub unsafe trait FromBytes
where
    Self: TransmuteFrom<[u8], { Assume::SAFETY }>
{}

pub trait IntoBytes
where
    [u8]: TransmuteFrom<Self>
{}
```


Where we can provide bare DSTs to `TransmuteFrom`. In such cases, the compiler will imagine a DST instantiated to be of sufficiently great length, and perform a transmutability analysis with that virtual type.

Fallible Transmutation

Before

```
fn u8_to_bool(src: u8) -> bool {
    assert!(src < 2);
    // SAFETY: We have checked that `src` is a bit-valid
    // instance of `bool`.
    unsafe {
        TransmuteFrom::<_, Assume::VALIDITY>::transmute(src)
    }
}
```

After (Speculative)

```
fn u8_to_bool(src: u8) -> Option<bool> {
    // SAFETY: No safety obligations!
    unsafe { TryTransmuteFrom::try_transmute(src) }
}
```

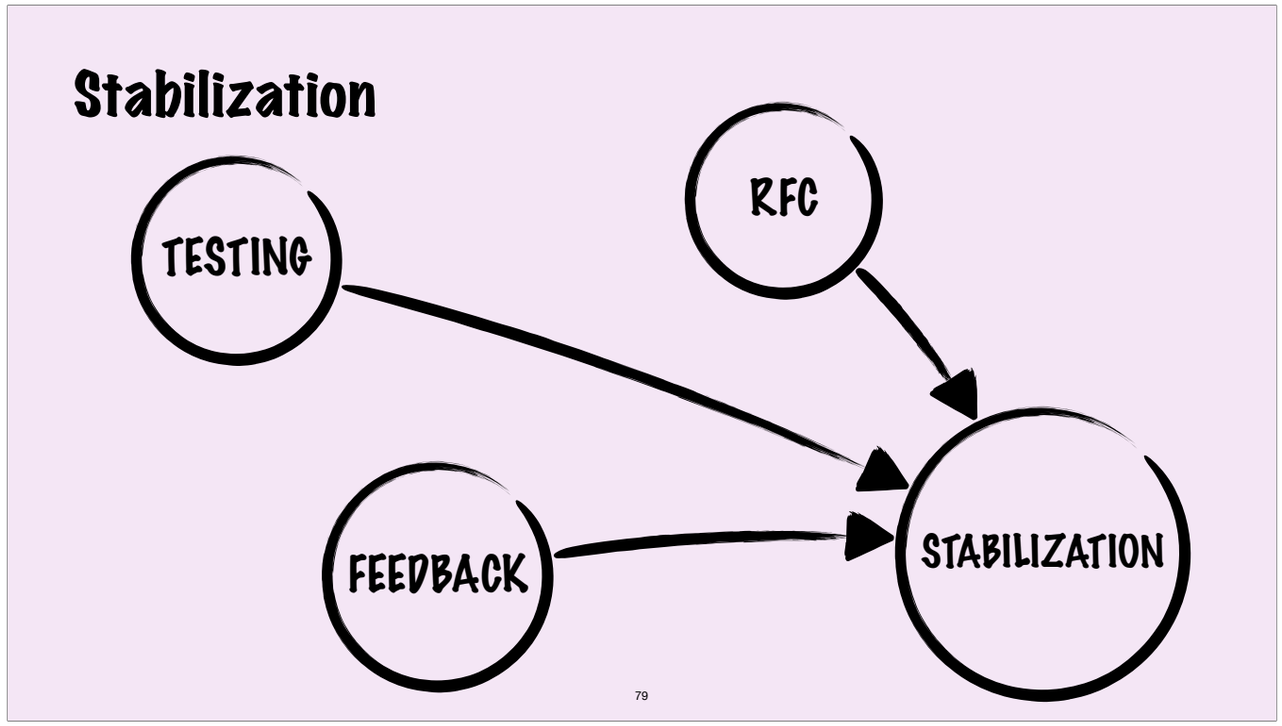

Next, is fallible transmutation. Sure, with `Assume` we can tell the compiler to assume we're taking care of doing certain checks. For example, with `Assume::VALIDITY`, the compiler will let us transmute a `u8` to a `bool`, but the onus is on us to correctly implement that runtime check on the source value.

But what if the compiler could do this for us, as well? Perhaps we could have a complementary `TryTransmuteTrait` with a `try_transmute` associated function that would automatically codegen the necessary runtime checks. This would further prevent accidental undefined behavior.

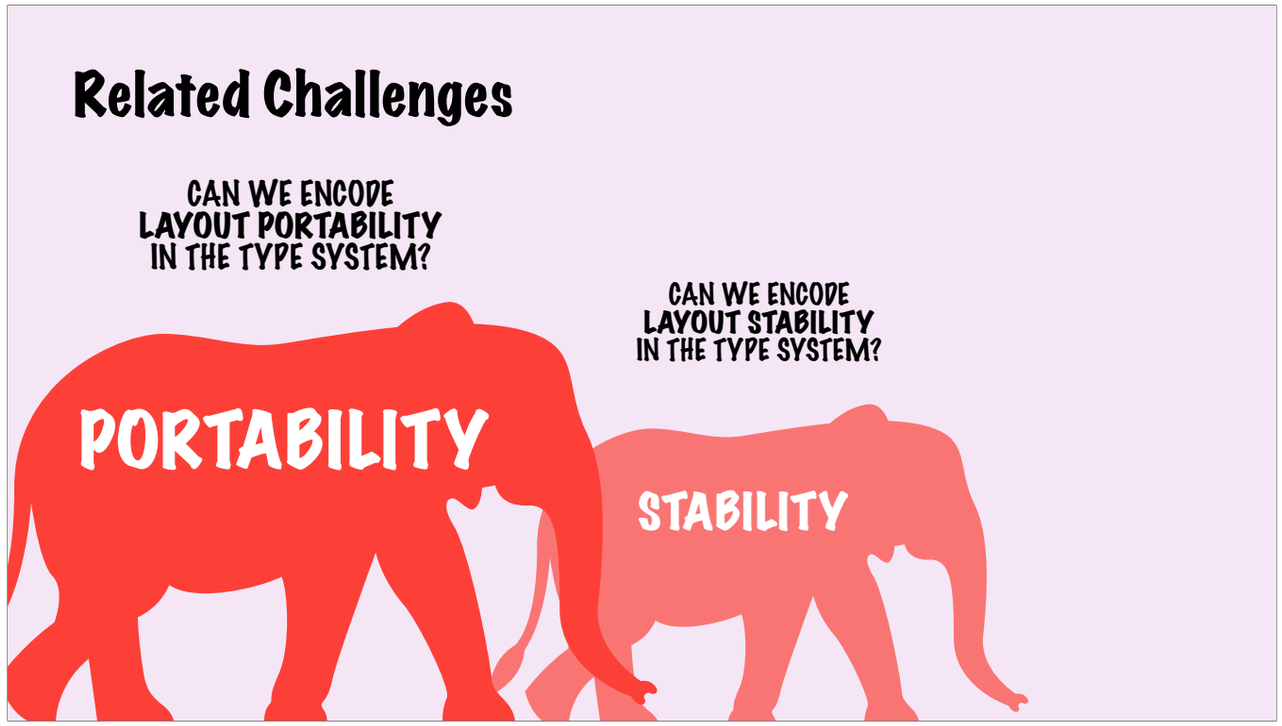
Critical Optimizations

- Avoid recursion.
- Dense representation of layout graphs.
- Run-length-encoding optimization for arrays.

And third, to make this production ready, we need to implement a variety of optimizations to our implementation. We need to avoid recursion, migrate to a denser memory representation of layouts, and implement a run-length-encoding optimization so we can support very large arrays efficiently.



Finally, we'll be able to look towards stabilization, which will require testing, feedback, and, of course an RFC. If all goes well, I can see us making that RFC in 2025.



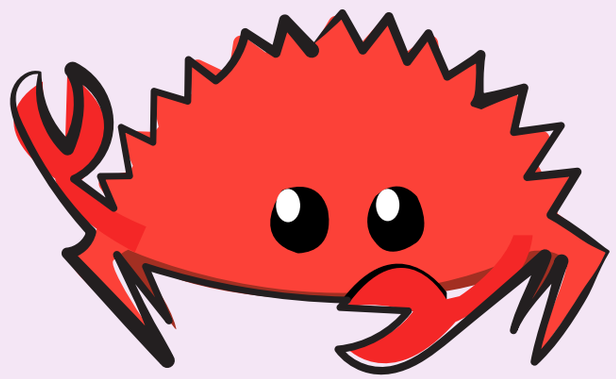
And even then, there will still be more work to be done. The two elephants in the room are portability and stability.

Rust provides very little in the way of layout portability. For example, the alignments of primitive integers vary between platforms, and this can have downstream effects on the validity of pointer casts. We have some ideas of how to solve this.

And, relatedly: It’d also be great if we could reflect SemVer guarantees about layouts and transmutability in the type system.

Special Thanks

- You! Yes, You!
- Ryan Levick
- Josh Liebow-Feeser
- Lokathor
- Eli Rosenthal
- Michael Goulet
- Oli Scherer
- Bryan Garza



If any of this happens, it will be thanks to the support of community members like you — and all the phenomenal collaborators who've helped us get this far. Thank you! I'll end this with a call for your help.

Safety Goggles for Alchemists

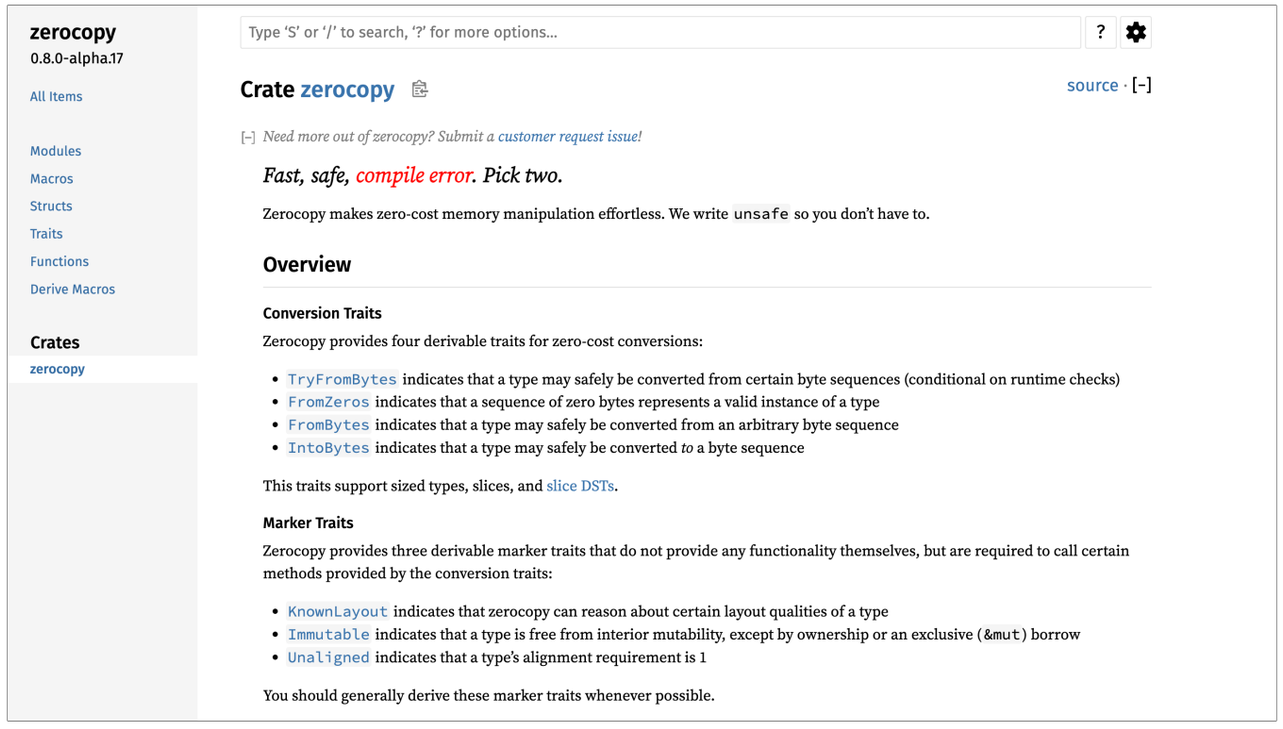
`#[feature(transmutability)]`

```
unsafe trait TransmuteFrom<Src: ?Sized, const ASSUME: Assume> {
    unsafe fn transmute(src: Src) -> Dst
    where
        Src: Sized,
        Self: Sized;
}
```

TRY IT OUT!

```
struct Assume {
    alignment: bool,
    lifetimes: bool,
    safety:    bool,
    validity:  bool,
}
```


The `TransmuteFrom` trait is available on nightly now, and it needs your review! We need to know whether it really covers as many use-cases as we think it does, and we want to discover what use-cases it doesn't cover.



Second, the zerocopy crate has been an invaluable sandbox for rapid experimentation outside the Rust compiler. We just cut a release-candidate of our biggest upgrade yet, featuring support for fallible transmutation, unsized types, and unsafe cells. Before we commit to releasing 0.8 for real, we need your eyes on it!

In short, go forth and transmute!