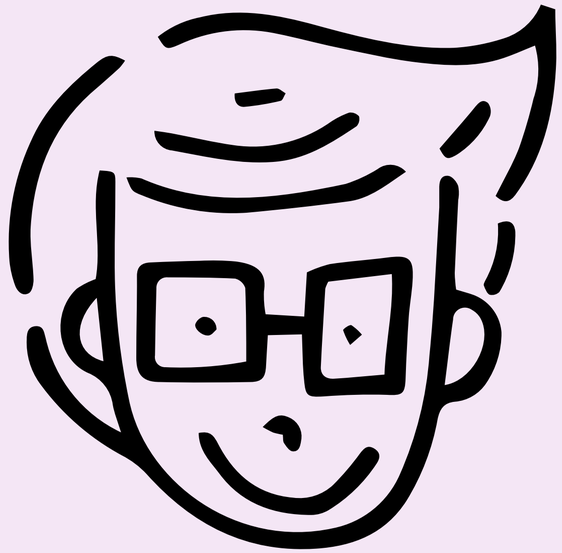
Safety Goggles *for* Alchemists

Jack Wrenn | RustConf 2024

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

History of Rust

Rust programming language

(a.k.a. “Project Servo”)

Technology from the past,
come to save the future
from itself.

Mozilla Annual Summit, July 2010

<graydon@mozilla.com>

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

History of Rust

Nicholas Flamel

1330–1418

History of Rust

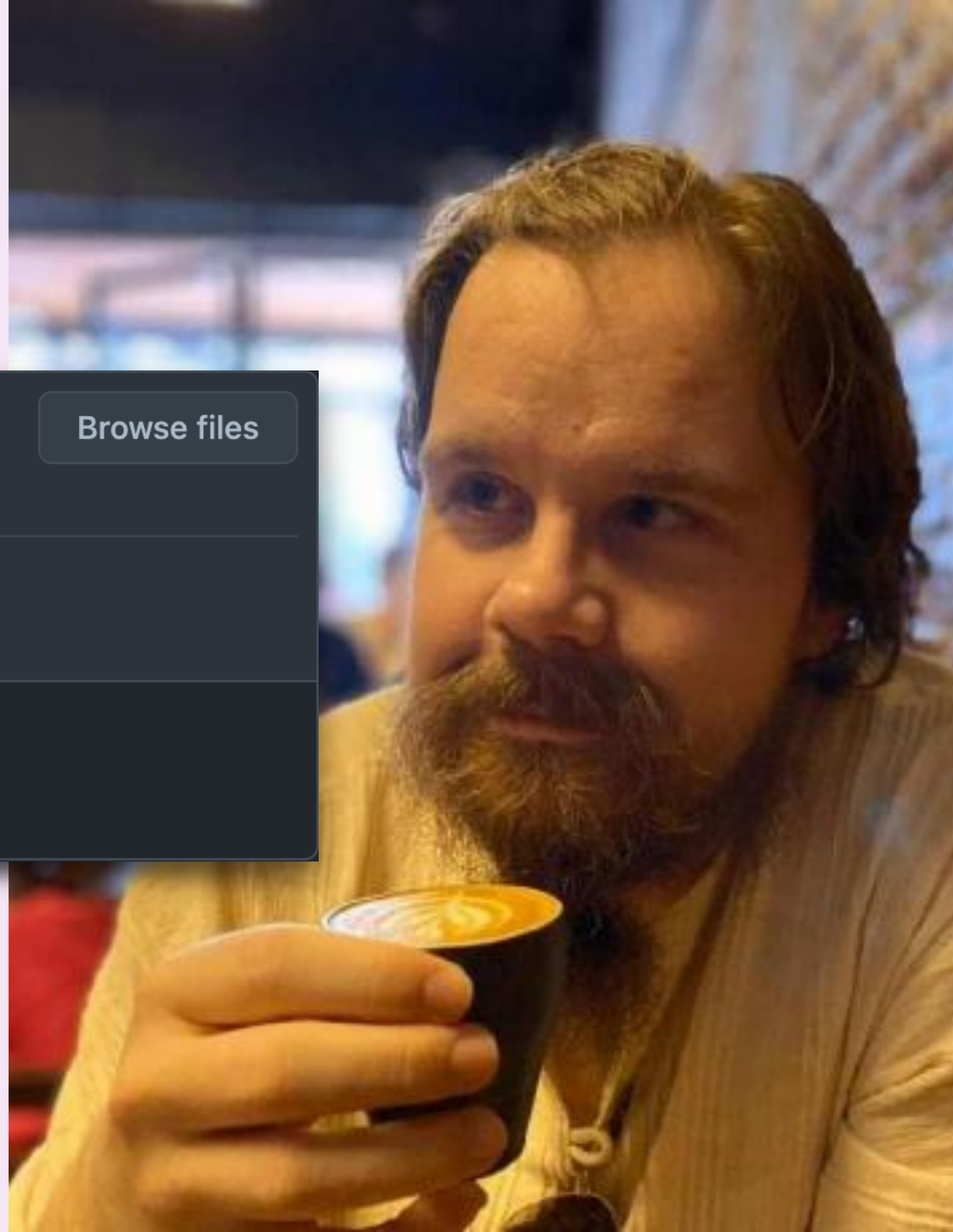
core: Add unsafe::transmute Browse files

Like reinterpret_cast + forget

master
release-0.7 ... 0.3

brson committed on Jun 8, 2012

1 parent 75d9ec1 commit f12adcb



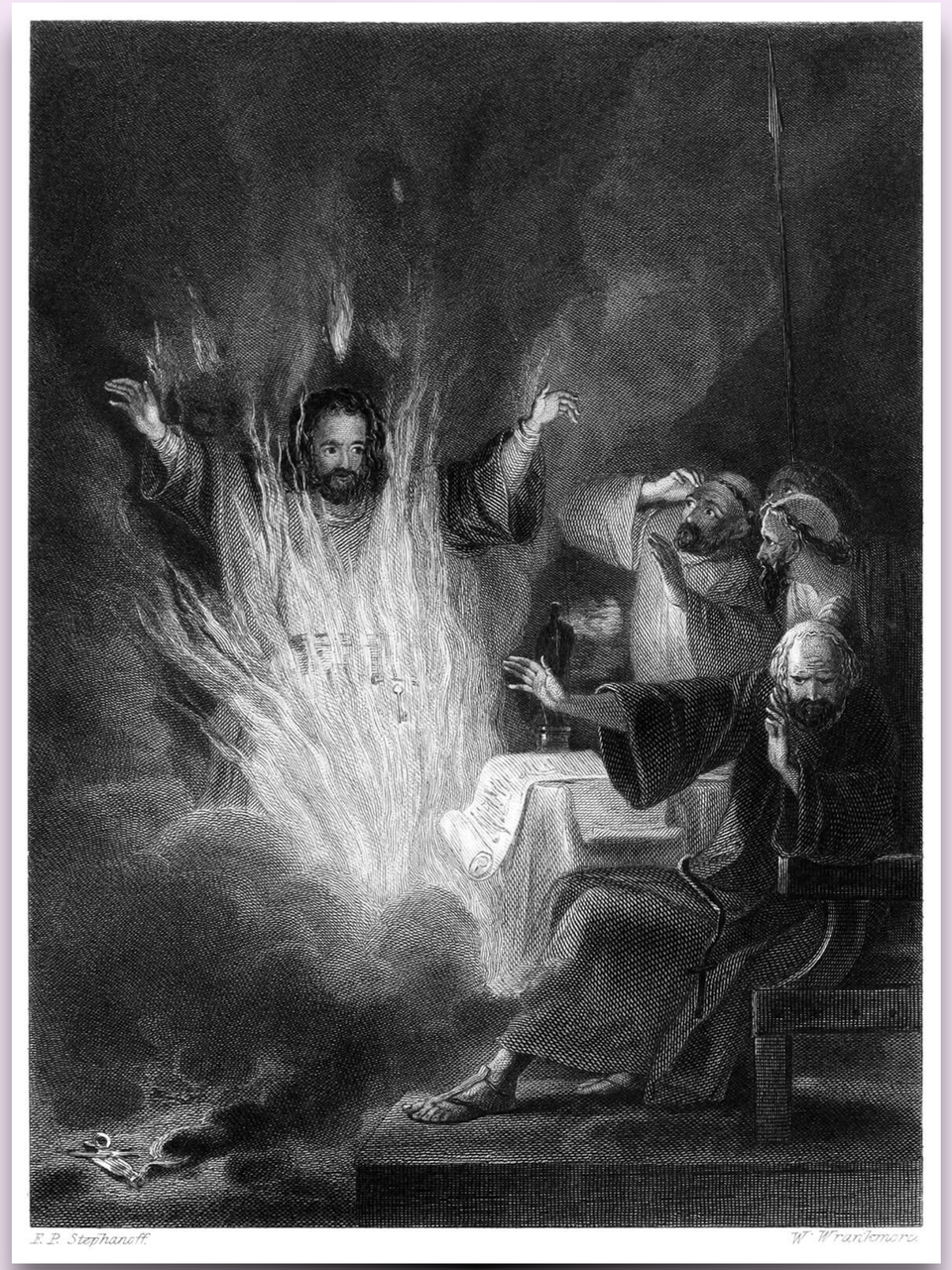
Alchemy in Rust

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

Alchemy in Rust

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

struct Lead;
struct Gold;

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

Risks



1. Bit Validity

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

2. Alignment

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

3. Lifetime Extension

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

4. Safety Invariants

Declaration

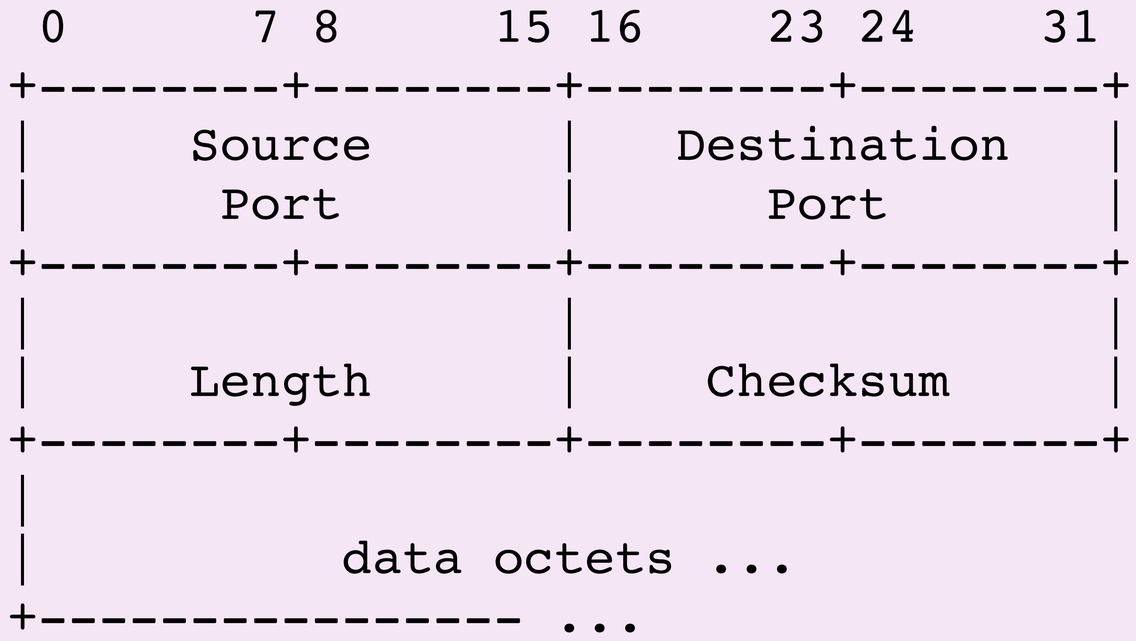
```
pub struct Even {
    // SAFETY: Always an even number!
    n: u8
}

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

Use

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

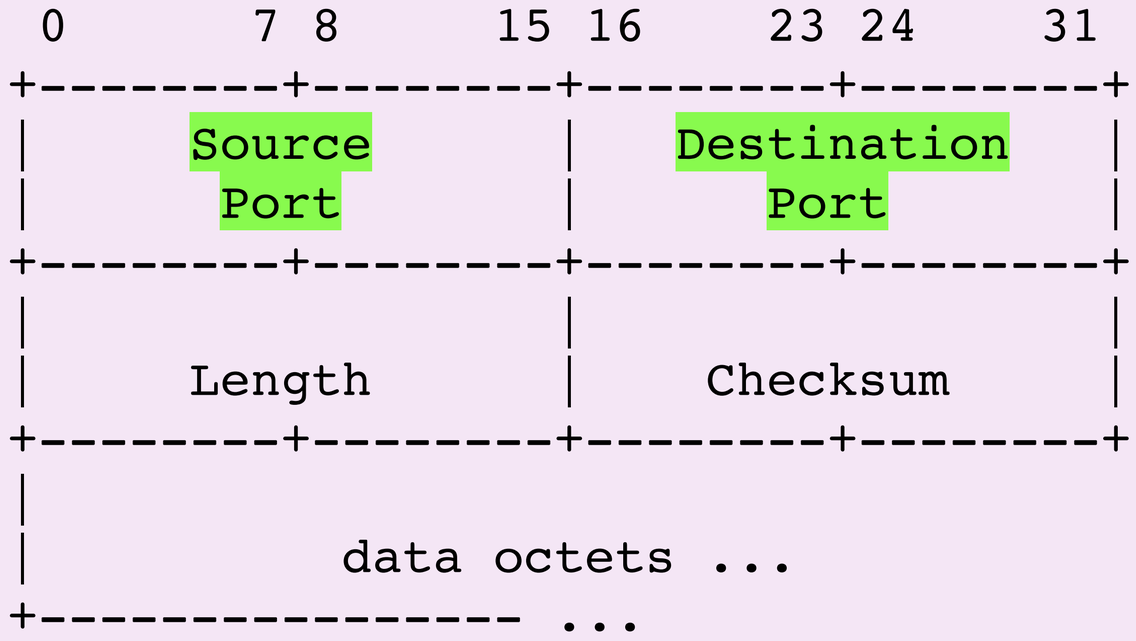
Rewards

Case Study: Packet Parsing



User Datagram Header Format

Case Study: Packet Parsing



User Datagram Header Format

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),
    })
}
```

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),
    })
}
```

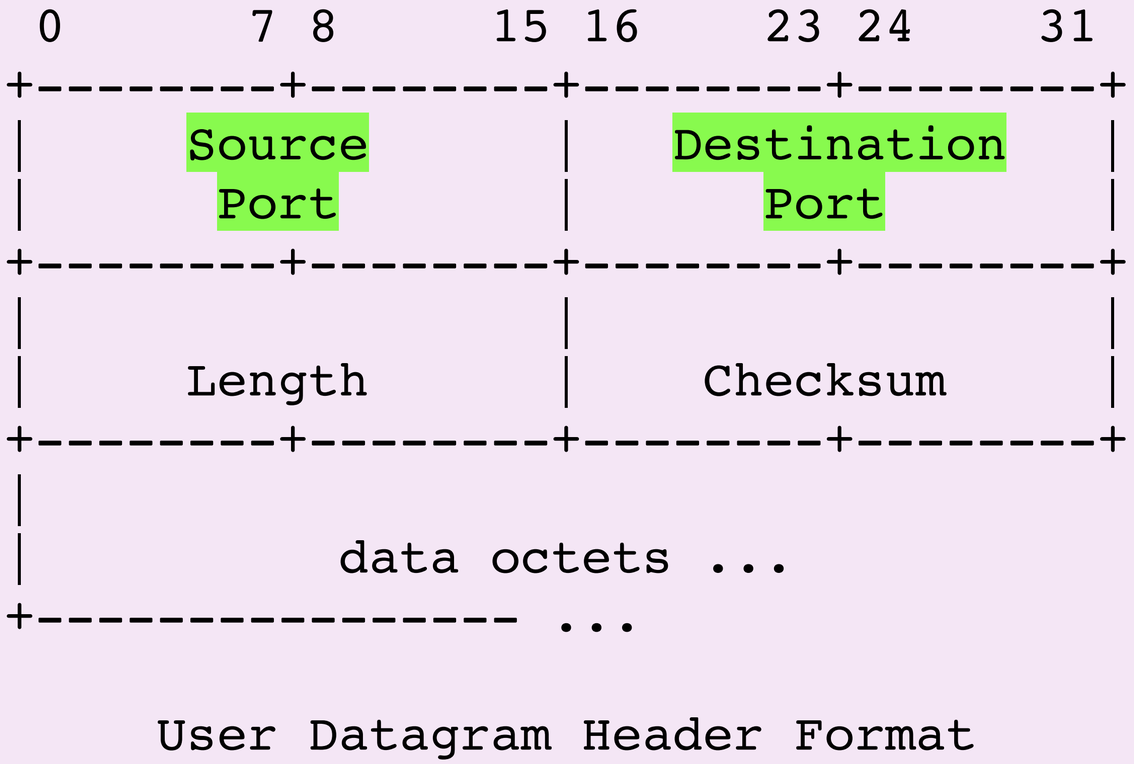
Case Study: Packet Parsing

```
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),
    })
}
```

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),
    })
}
```

Case Study: Packet Parsing



User Datagram Header Format

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

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) })
}
```

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) })
}
```

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) })
}
```

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
```

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) })
}
```

Transmutation Crates

Crate bytemuck

[−] This crate gives small utilities for casting between plain data types.

Basics

Data comes in five basic forms in Rust, so we have five basic casting functions:

- `T` uses `cast`
- `&T` uses `cast_ref`
- `&mut T` uses `cast_mut`
- `&[T]` uses `cast_slice`
- `&mut [T]` uses `cast_slice_mut`

Depending on the function, the `NoUninit` and/or `AnyBitPattern` traits are used to maintain memory safety.

Historical Note: When the crate first started the `Pod` trait was used instead, and so you may hear people refer to that, but it has the strongest requirements and people eventually wanted the more fine-grained system, so here we are. All types that impl `Pod` have a blanket impl to also support `NoUninit` and `AnyBitPattern`. The traits unfortunately do not have a perfectly clean hierarchy for semver reasons.

Failures

Some casts will never fail, and other casts might fail.

- `cast::<u32, f32>` always works (and `f32::from_bits`).
- `cast_ref::<[u8; 4], u32>` might fail if the specific array reference given at runtime doesn't have alignment 4.

In addition to the "normal" forms of each function, which will panic on invalid input, there's also `try_` versions which will return a `Result`.

If you would like to statically ensure that a cast will work at runtime you can use the `must_cast` crate feature and the `must`

Crate zerocopy

[−] *Need more out of zerocopy? Submit a customer request issue!*

Fast, safe, compile error. Pick two.

Zerocopy makes zero-cost memory manipulation effortless. We write `unsafe` so you don’t have to.

Overview

Conversion Traits

Zerocopy provides four derivable traits for zero-cost conversions:

- `TryFromBytes` indicates that a type may safely be converted from certain byte sequences (conditional on runtime checks)
- `FromZeros` indicates that a sequence of zero bytes represents a valid instance of a type
- `FromBytes` indicates that a type may safely be converted from an arbitrary byte sequence
- `IntoBytes` indicates that a type may safely be converted *to* a byte sequence

This traits support sized types, slices, and slice DSTs.

Marker Traits

Zerocopy provides three derivable marker traits that do not provide any functionality themselves, but are required to call certain methods provided by the conversion traits:

- `KnownLayout` indicates that zerocopy can reason about certain layout qualities of a type
- `Immutable` indicates that a type is free from interior mutability, except by ownership or an exclusive (`&mut`) borrow
- `Unaligned` indicates that a type’s alignment requirement is 1

You should generally derive these marker traits whenever possible.

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

Inside Safe Transmutation Crates

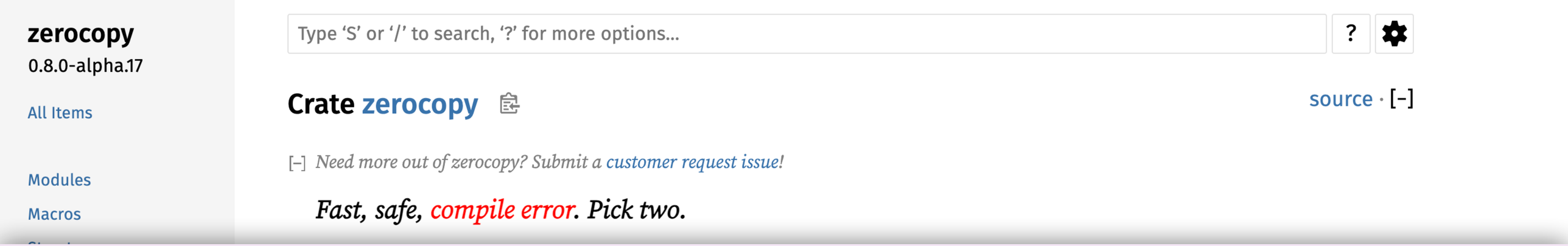
Marker Traits and Inductive Reasoning

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

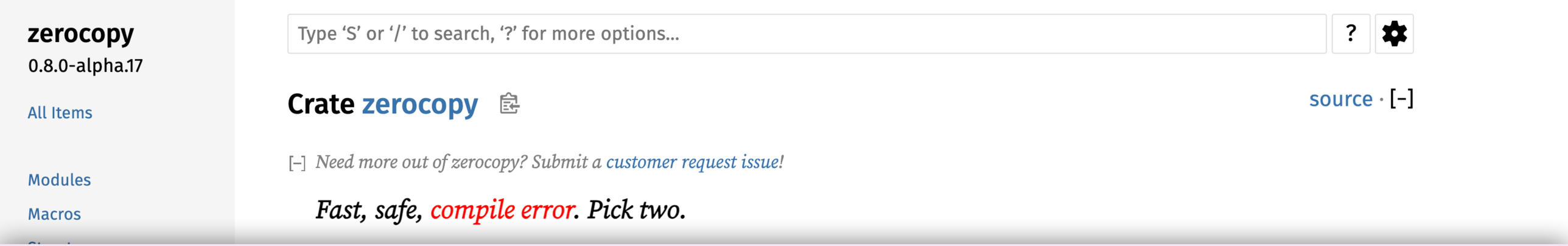
unsafe marker trait

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

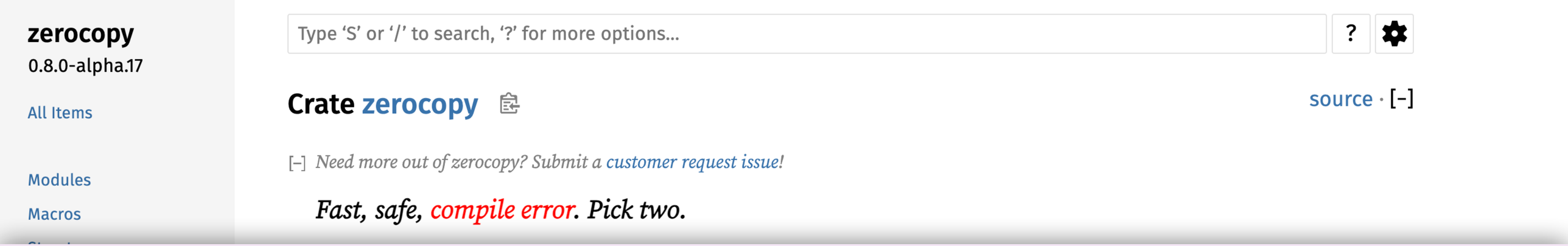
inductive, compound implementations




```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> {
   …
}
```




```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> {
    …
}
```




```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()
}
```

Netstack3 - A Fuchsia owned rust based netstack

- Authors: ghanan@google.com, asafi@google.com
- Project lead: tamird@google.com
- Area(s): Connectivity

Problem statement

The current netstack is written in Go which is not an approved language (and is garbage collected) and is owned by the gVisor team who have different priorities, use cases and design goals than Fuchsia. The gVisor netstack was not originally designed to run on real devices, operate as a router or support dynamic configurations; Fuchsia on the other hand does run on real devices, operates as a router and depends on dynamic configurations (gVisor originally didn't expect addresses, routes, network interfaces or their link status to change at runtime).

Solution statement

By developing a netstack that the Fuchsia team owns, we can design with Fuchsia's goals in mind without having to depend on another team and work around their use cases and restrictions. The Fuchsia Netstack team will design and implement a Rust-based netstack that achieves functional parity with the existing netstack, while leveraging the type and memory safety of rust.

zerocopy
0.8.0-alpha.17

All Items

Modules
Macros
Structs
Traits
Functions
Derive Macros

Crates
zerocopy

Type ‘S’ or ‘/’ to search, ‘?’ for more options...

Crate zerocopy

source · [−]

[−] *Need more out of zerocopy? Submit a customer request issue!*

Fast, safe, compile error. Pick two.

Zerocopy makes zero-cost memory manipulation effortless. We write `unsafe` so you don’t have to.

Overview

14,000

Lines of Code, Including Comments

Conversion Traits

Zerocopy provides fo erivab -cost ers :

- `TryFromBytes` i cates that a typ ay sa ly b ed fro byte s (conditional on runtime checks)
- `FromZeros` indicates that a sequenc f ze ytes represents a valid instance of a type
- `FromBytes` indicates that a type may safely be converted from an arbitrary byte sequence
- `IntoBytes` indicates that a type may safely be converted *to* a byte sequence

This traits support sized types, slices, and slice DSTs.

Marker Traits

Zerocopy provides three derivable marker traits that do not provide any functionality themselves, but are required to call certain methods provided by the conversion traits:

- `KnownLayout` indicates that zerocopy can reason about certain layout qualities of a type
- `Immutable` indicates that a type is free from interior mutability, except by ownership or an exclusive (`&mut`) borrow
- `Unaligned` indicates that a type’s alignment requirement is 1

You should generally derive these marker traits whenever possible.

Crate zerocopy

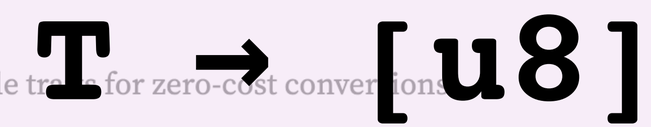
[−] *Need more out of zerocopy? Submit a customer request issue!*

Fast, safe, compile error. Pick two.

Zerocopy makes zero-cost memory manipulation effortless. We write `unsafe` so you don't have to.

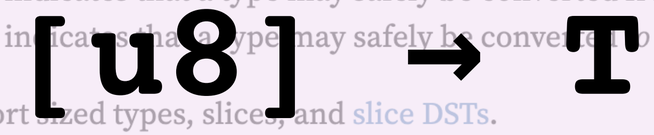
Overview

Conversion Traits



Zerocopy provides four derivable traits for zero-cost conversions:

- `TryFromBytes` indicates that a type may safely be converted from certain byte sequences (conditional on runtime checks)
- `FromZeros` indicates that a sequence of zero bytes represents a valid instance of a type
- `FromBytes` indicates that a type may safely be converted from an arbitrary byte sequence
- `IntoBytes` indicates that a type may safely be converted to a byte sequence



This traits support sized types, slices, and slice DSTs.

Marker Traits

Zerocopy provides three derivable marker traits that do not provide any functionality themselves, but are required to call certain methods provided by the conversion traits:

- `KnownLayout` indicates that zerocopy can reason about certain layout qualities of a type
- `Immutable` indicates that a type is free from interior mutability, except by ownership or an exclusive (`&mut`) borrow
- `Unaligned` indicates that a type's alignment requirement is 1

You should generally derive these marker traits whenever possible.

Project Safe Transmute

RFC 2835

Theory of Alchemy

Theory of Alchemy

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

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)
```

Theory of Alchemy

Observation: Sets are Computationally Expensive

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

Theory of Alchemy

Observation: Types May Not Be Finite

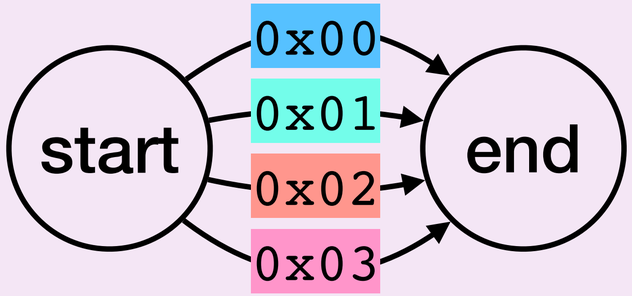
```
struct LinkedList<T> {
    head: T,
    tail: Option<Box<T>>,
}
```

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

Theory of Alchemy

Observation: Types are Automata

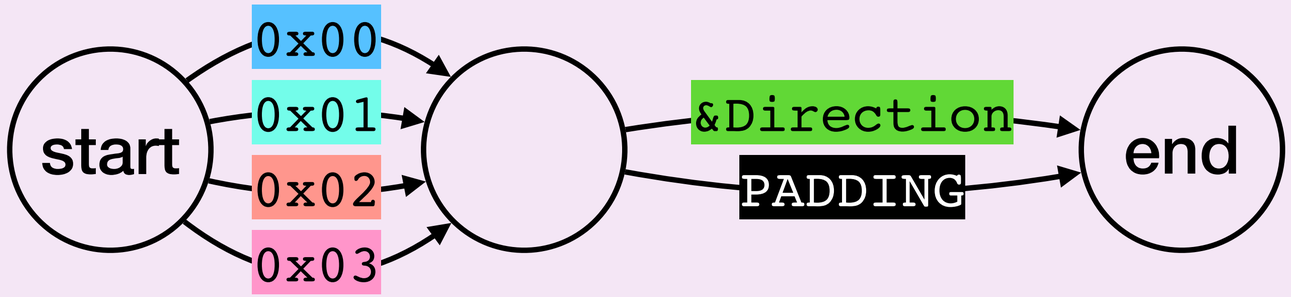
```
#[repr(u8)]
enum Direction {
    North,
    East,
    South,
    West
}
```



Theory of Alchemy

Observation: Automata Are Expressive

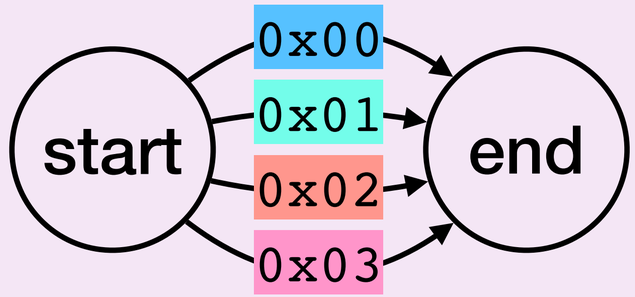
```
#[repr(C)]
struct LinkedList {
    head: Direction,
    tail: Option<Box<Direction>>,
}

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



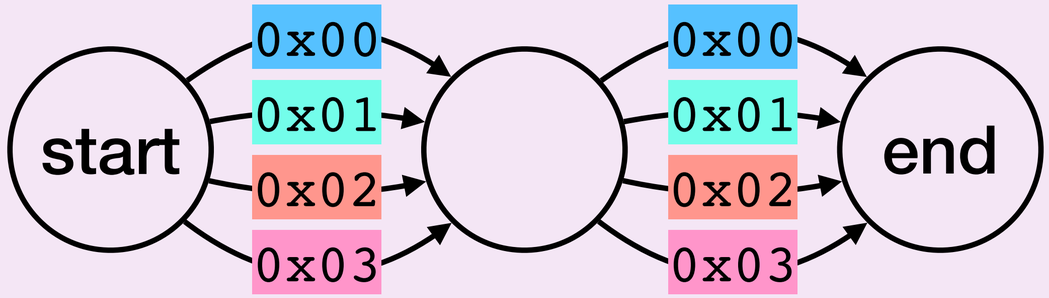
Theory of Alchemy

Observation: Automata Scale Gracefully

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




```
[Direction; 2]
```

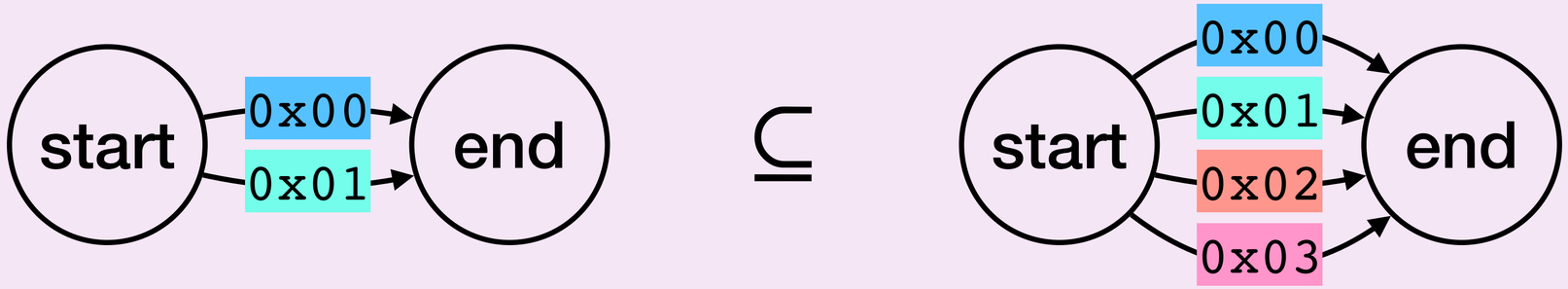


16 possible values

only 8 edges

Theory of Alchemy

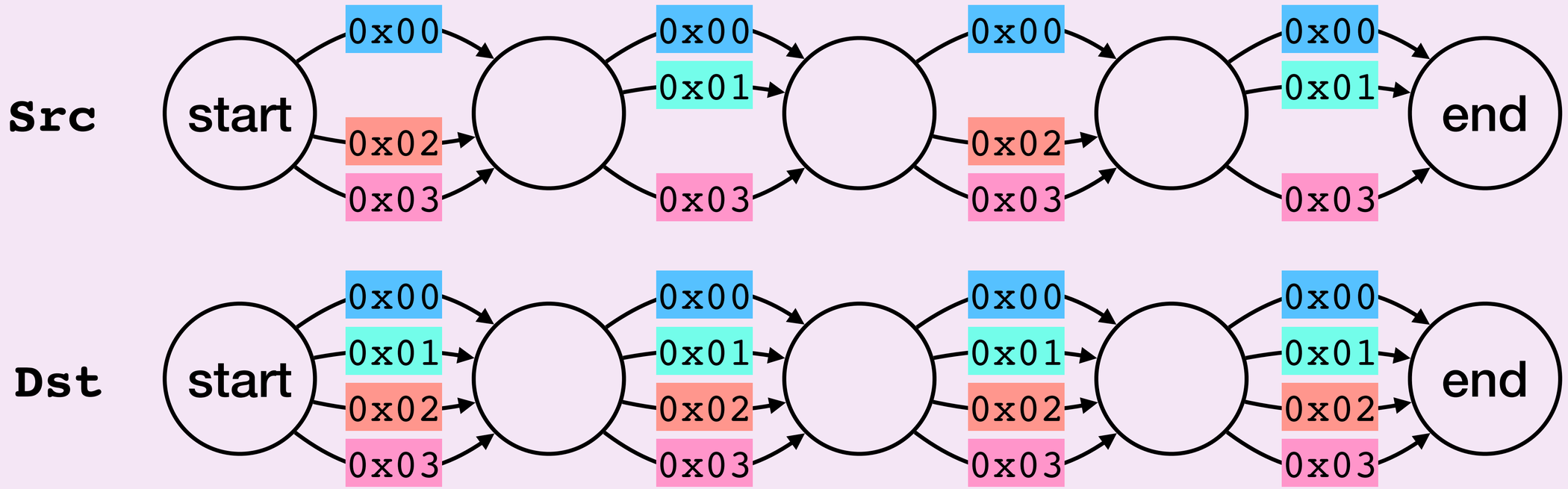
Observation: Transmutation with Automata is Simple



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

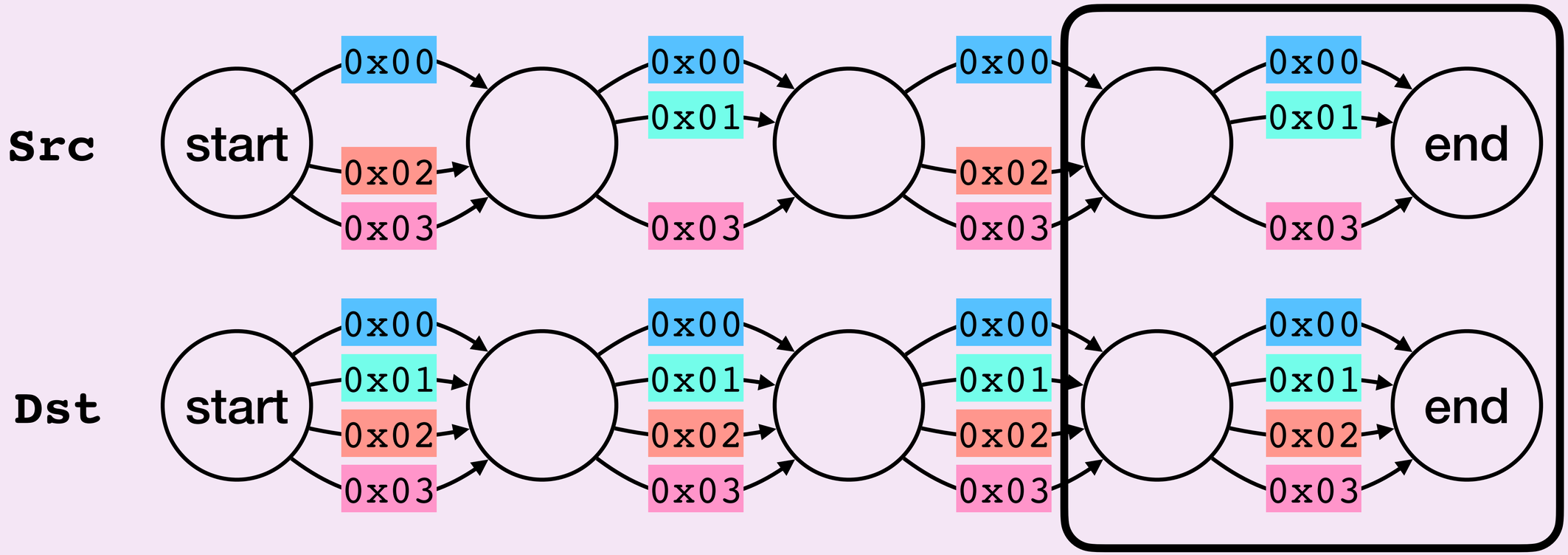
Theory of Alchemy

Observation: Transmutation with Automata is Efficient



Theory of Alchemy

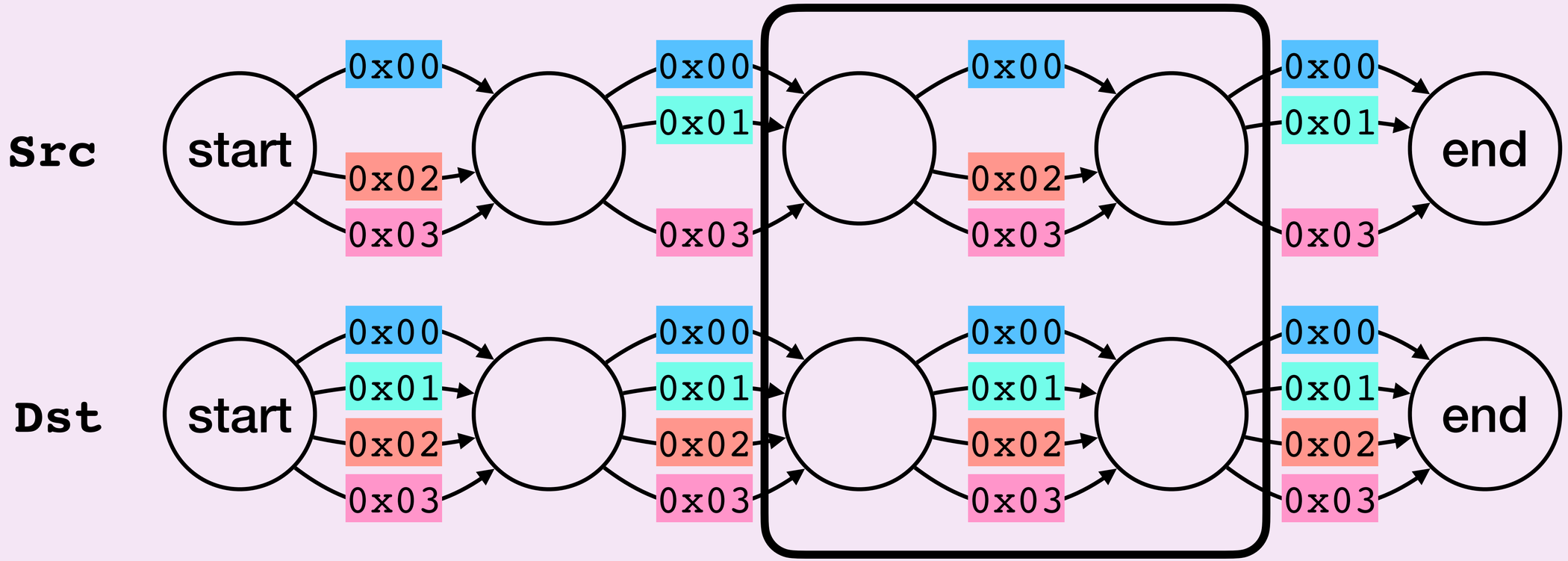
Observation: Transmutation with Automata is Efficient



*this is a sketch

Theory of Alchemy

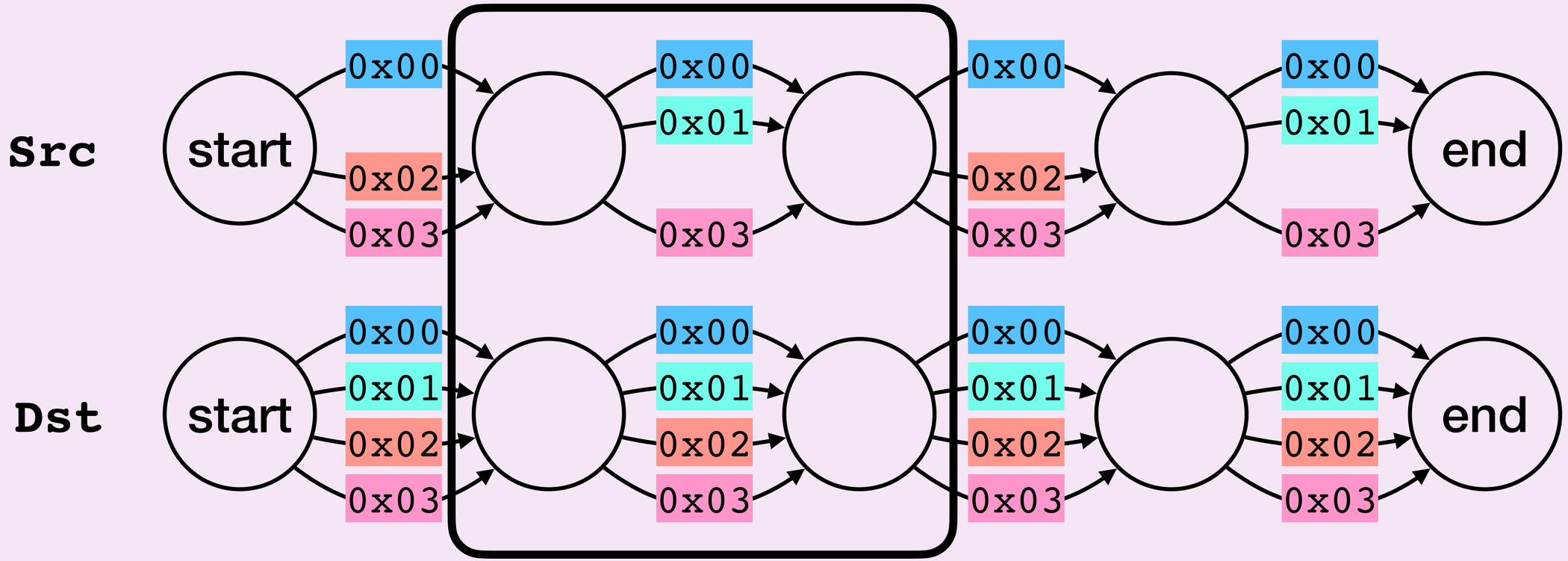
Observation: Transmutation with Automata is Efficient



*this is a sketch

Theory of Alchemy

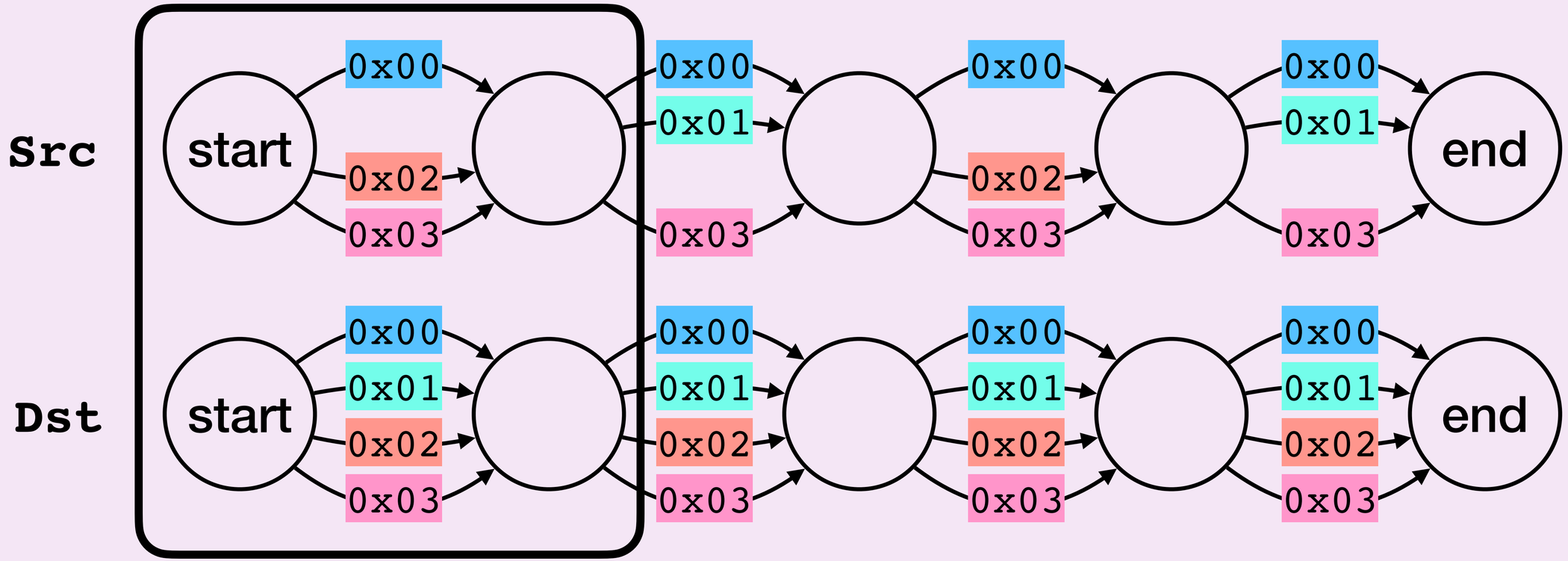
Observation: Transmutation with Automata is Efficient



*this is a sketch

Theory of Alchemy

Observation: Transmutation with Automata is Efficient



*this is a sketch

Dst: TransmuteFrom<Src>

MCP411 • #[feature(transmutability)]

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

Safe Alchemy

From Comments to Compile Errors

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
```

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
```

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`
```

4. Safety Invariants

Upstream

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

Downstream

```
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
```

Safer Alchemy

TransmuteFrom & Assume

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

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

Assume lets you configure compile-time safety checks

`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)
    }
}
```

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)
    }
}
```

`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)
    }
}
```

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;
}
```

AVAILABLE NOW!

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

Safety Goggles for Transmute

```
#[feature(transmutability)]

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

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

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

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)
}
```

Safety Goggles for Abstraction

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

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

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
{}
```

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
{}
```

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>
{}
```

zerocopy
0.8.0-alpha.17

All Items

Modules
Macros
Structs
Traits
Functions
Derive Macros

Crates
zerocopy

Type ‘S’ or ‘/’ to search, ‘?’ for more options...

Crate zerocopy

source · [−]

[−] *Need more out of zerocopy? Submit a customer request issue!*

Fast, safe, compile error. Pick two.

Zerocopy makes zero-cost memory manipulation effortless. We write `unsafe` so you don’t have to.

Overview

14,000

Lines of Code

Conversion Traits

Zerocopy provides fo erivab -cost ers :

- `TryFromBytes` i cates that a typ ay sa ly b ed fro byte s (conditional on runtime checks)
- `FromZeros` indicates that a sequenc f ze ytes represents a valid instance of a type
- `FromBytes` indicates that a type may safely be converted from an arbitrary byte sequence
- `IntoBytes` indicates that a type may safely be converted to a byte sequence

This traits support sized types, slices, and slice DSTs.

Marker Traits

Zerocopy provides three derivable marker traits that do not provide any functionality themselves, but are required to call certain methods provided by the conversion traits:

- `KnownLayout` indicates that zerocopy can reason about certain layout qualities of a type
- `Immutable` indicates that a type is free from interior mutability, except by ownership or an exclusive (`&mut`) borrow
- `Unaligned` indicates that a type’s alignment requirement is 1

You should generally derive these marker traits whenever possible.

zerocopy
0.8.0-alpha.17

All Items

Modules
Macros
Structs
Traits
Functions
Derive Macros

Crates
zerocopy

Type ‘S’ or ‘/’ to search, ‘?’ for more options...

Crate zerocopy

source · [−]

[−] *Need more out of zerocopy? Submit a customer request issue!*

Fast, safe, compile error. Pick two.

Zerocopy makes zero-cost memory manipulation effortless. We write `unsafe` so you don’t have to.

Overview

1,400
Lines of Code

Conversion Traits

Zerocopy provides four derivable [illegible] conversion

- `TryFromBytes` indicates that a type may safely be [illegible] from certain byte sequences (conditional on runtime checks)
- `FromZeros` indicates that a sequence of zero bytes represents a valid instance of a type
- `FromBytes` indicates that a type may safely be converted from an arbitrary byte sequence
- `IntoBytes` indicates that a type may safely be converted to a byte sequence

This traits support sized types, slices, and slice DSTs.

Marker Traits

Zerocopy provides three derivable marker traits that do not provide any functionality themselves, but are required to call certain methods provided by the conversion traits:

- `KnownLayout` indicates that zerocopy can reason about certain layout qualities of a type
- `Immutable` indicates that a type is free from interior mutability, except by ownership or an exclusive (`&mut`) borrow
- `Unaligned` indicates that a type’s alignment requirement is 1

You should generally derive these marker traits whenever possible.

Future Outlook

We need your help!

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>
{}
```

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>
{}
```

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)
    }
}
```

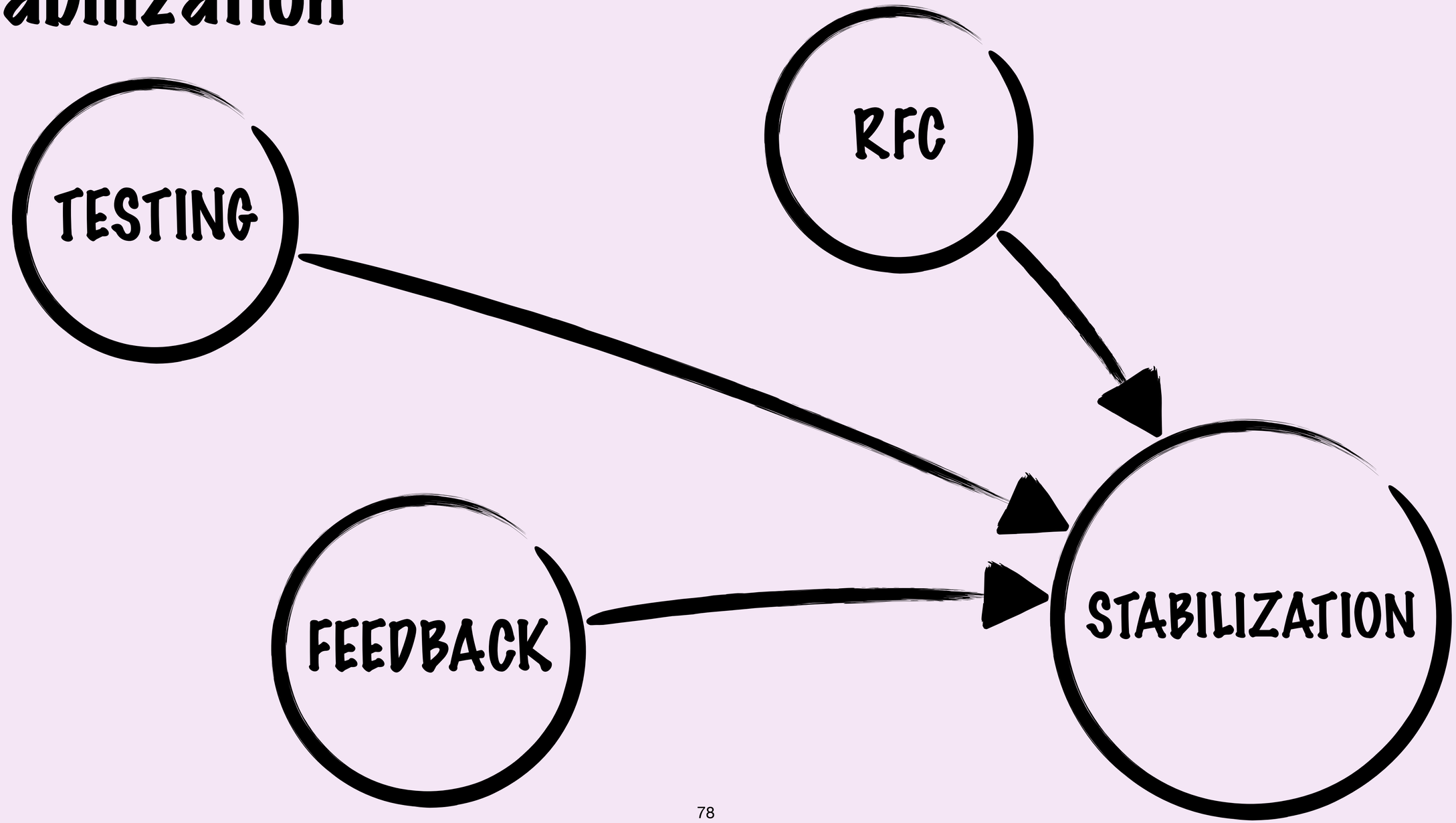
After (Speculative)

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

Critical Optimizations

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

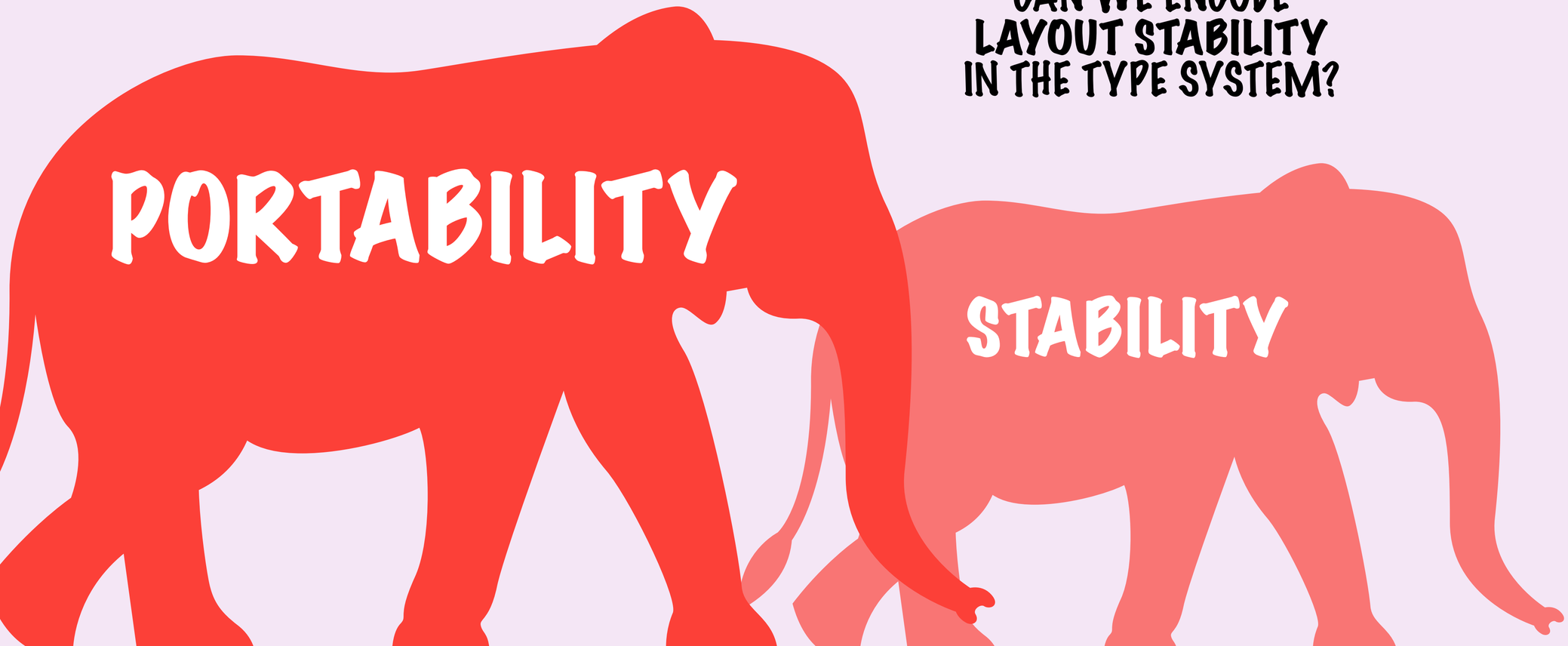
Stabilization



Related Challenges

CAN WE ENCODE
LAYOUT PORTABILITY
IN THE TYPE SYSTEM?

CAN WE ENCODE
LAYOUT STABILITY
IN THE TYPE SYSTEM?



Special Thanks

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