how to stop fighting with Coherence and start writing Context-Generic trait impls

by Soares Chen

TENSORDYNE

Soares Chen

- Software Engineer at Tensordyne
  - Building next generation AI inference infrastructure in Rust
- Rust, Haskell, JavaScript, etc
- Functional programming & programming language theory
- Creator of Context-Generic Programming

Agenda

- Overview of Rust trait system
  - Problems with Coherence
- Potential Solutions
- Context-Generic Programming (CGP)

Rust Traits

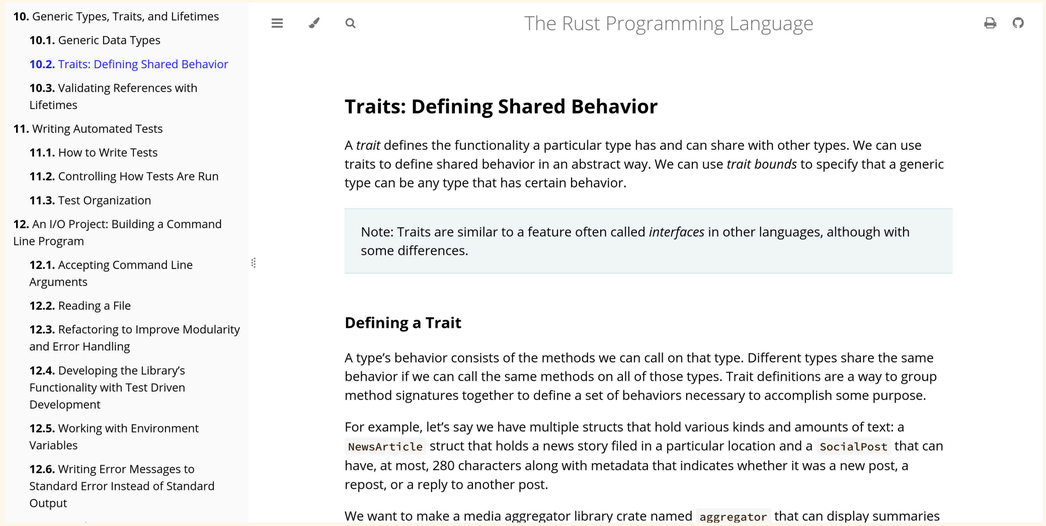
```rust
pub trait Display {
    fn fmt(&self, f: &mut Formatter<'_>) ->
        Result<(), Error>;
}
```

```rust
pub trait From<T>: Sized {
    fn from(value: T) -> Self;
}
```

```rust
pub trait Iterator {
    type Item;

    fn next(&mut self) -> Option<Self::Item>;
    ...
}
```

```rust
pub trait Serialize {
    fn serialize<S>(&self, serializer: S) ->
        Result<S::Ok, S::Error>
    where S: Serializer;
}
```

What are Traits?



Traits: Defining Shared Behavior

A *trait* defines the functionality a particular type has and can share with other types. We can use traits to define shared behavior in an abstract way. We can use *trait bounds* to specify that a generic type can be any type that has certain behavior.

> Note: Traits are similar to a feature often called *interfaces* in other languages, although with some differences.

Defining a Trait

A type's behavior consists of the methods we can call on that type. Different types share the same behavior if we can call the same methods on all of those types. Trait definitions are a way to group method signatures together to define a set of behaviors necessary to accomplish some purpose.

For example, let's say we have multiple structs that hold various kinds and amounts of text: a `NewsArticle` struct that holds a news story filed in a particular location and a `SocialPost` that can have, at most, 280 characters along with metadata that indicates whether it was a new post, a repost, or a reply to another post.

We want to make a media aggregator library crate named `aggregator` that can display summaries

define shared behavior
in an abstract way

Implement

↕

Trait

↕

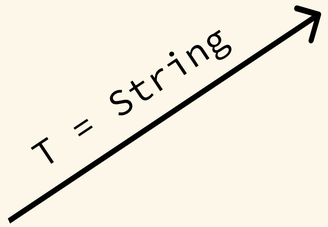
Use

similar to interfaces,
but with some differences

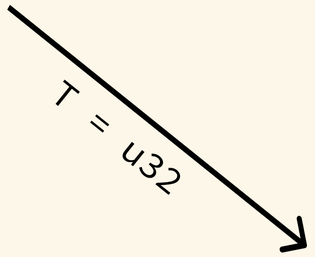
a generic type can be any type
that has certain behavior

Why Generics

```rust
fn greet<T: Display>(value: &T) {
    println!("Hello {value}")
}
```




```rust
fn greet(value: &String) {
    println!("Hello {value}")
}
```




```rust
fn greet(value: &u32) {
    println!("Hello {value}")
}
```

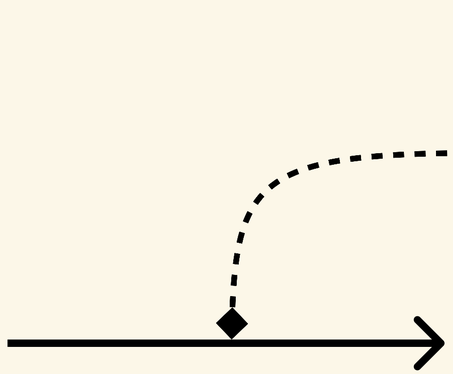
Monomorphization

Implementing Traits

```rust
pub struct Person {
    pub first_name: String,
    pub last_name: String,
}
```

```rust
impl Display for Person {
    fn fmt(&self, f: &mut Formatter<'_>) -> Result<(), Error> {
        write!(f, "{} {}", self.first_name, self.last_name)
    }
}
```

```rust
fn greet<T: Display>(value: &T) {
    println!("Hello {value}")
}
```




```rust
fn greet(value: &Person) {
    println!("Hello {value}")
}
```

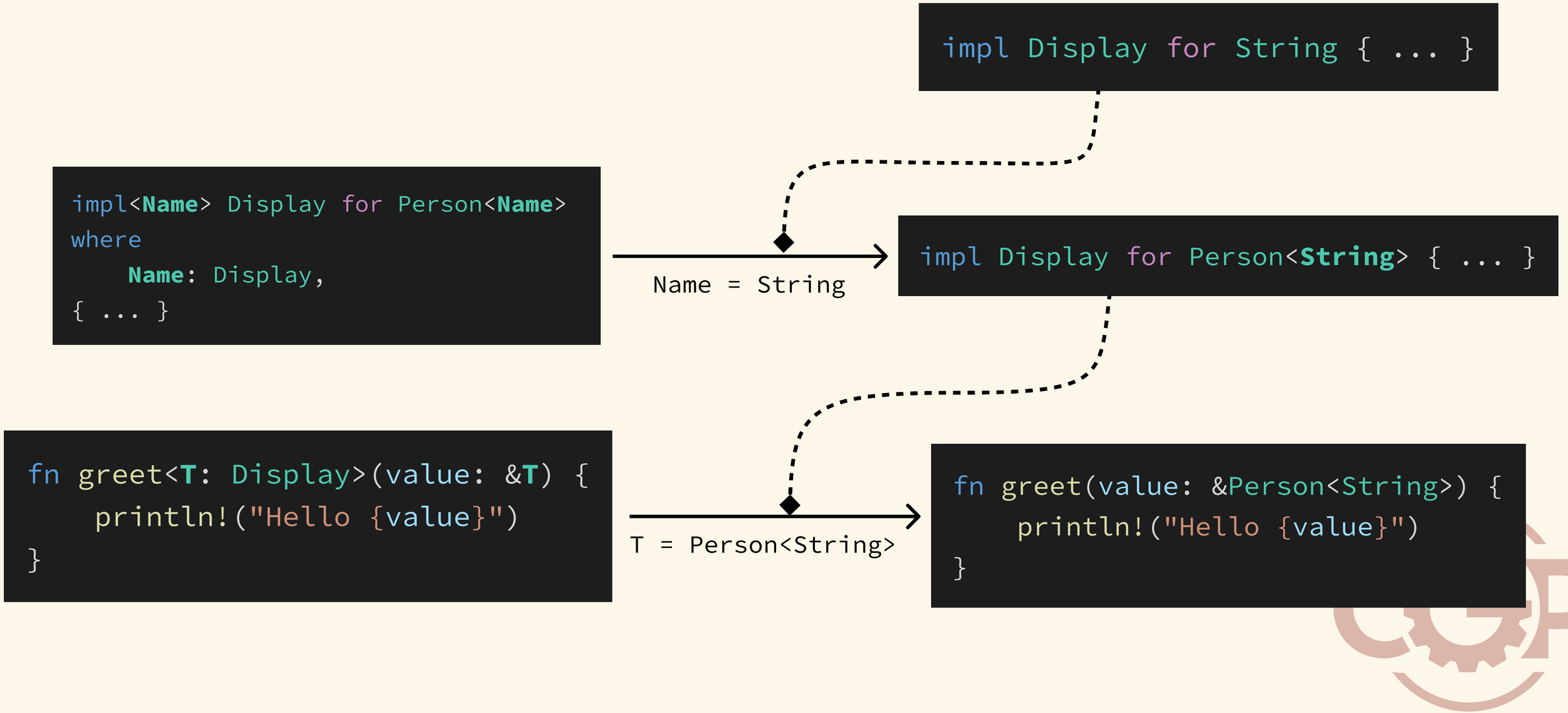
Generic Trait Implementations

```rust
pub struct Person<Name> {
    pub first_name: Name,
    pub last_name: Name,
}
```

Uses generics & implement traits generically

```rust
impl<Name> Display for Person<Name>
where
    Name: Display,
{
    fn fmt(
        &self,
        f: &mut Formatter<'_>,
    ) -> Result<(), Error> {
        write!(f, "{} {}",
            self.first_name,
            self.last_name,
        )
    }
}
```

Generic Instance Lookup



Dependency Injection with Rust Traits

No requirement of Display on Name here

```rust
pub struct Person<Name> {
    pub first_name: Name,
    pub last_name: Name,
}
```

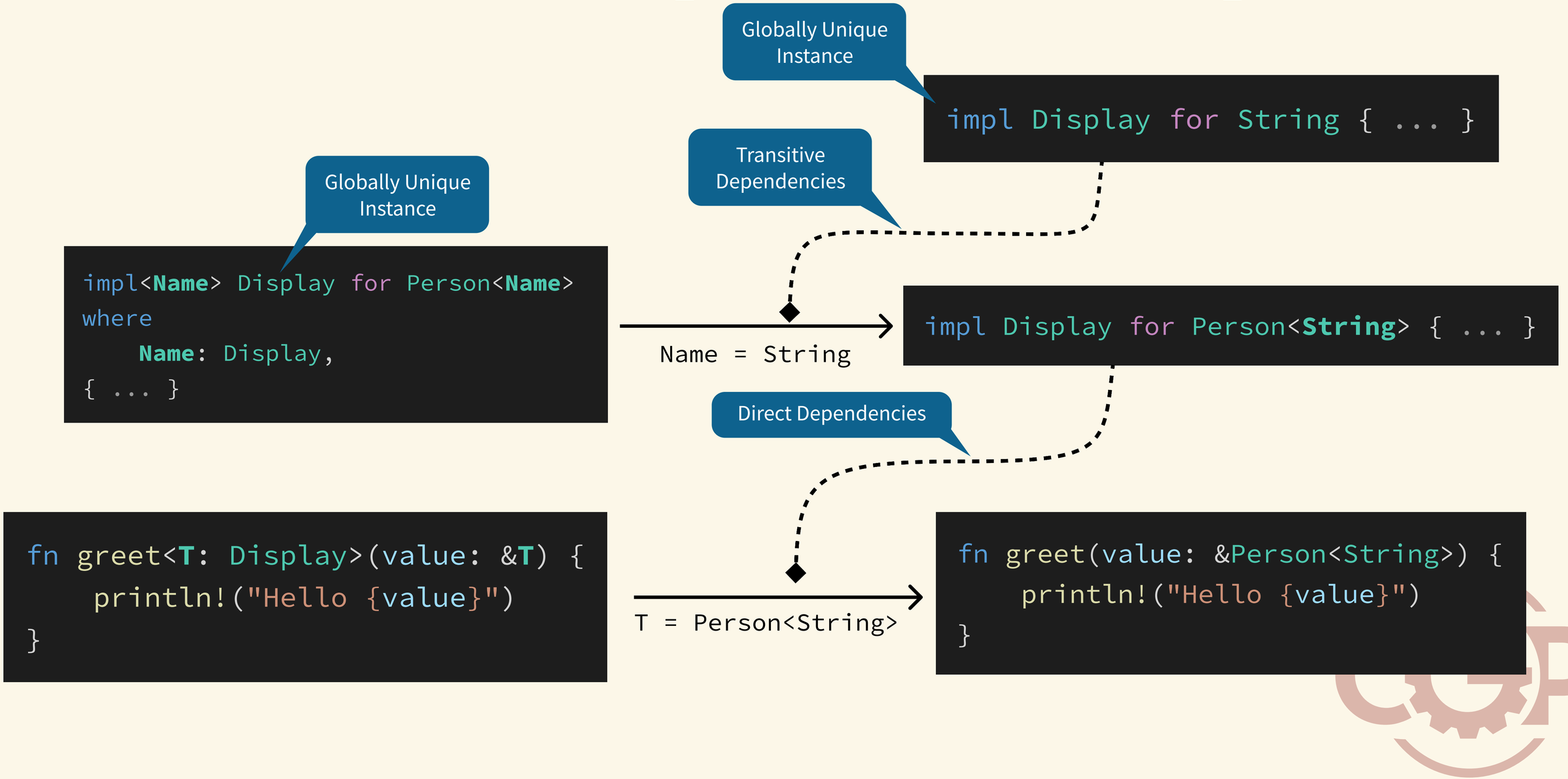
No requirement of Display on Name here

```rust
pub trait Display {
    fn fmt(&self, f: &mut Formatter<'_>) ->
        Result<(), Error>;
}
```

Hidden Dependencies

```rust
impl<Name> Display for Person<Name>
where
    Name: Display,
{
    fn fmt(
        &self,
        f: &mut Formatter<'_>,
    ) -> Result<(), Error> {
        write!(f, "{} {}",
            self.first_name,
            self.last_name,
        )
    }
}
```

Transitive Dependencies Lookup



The Coherence Problem

Dependency lookup must resolve to a globally unique instance

Global uniqueness is enforced by two rules:

❌ No Overlapping Instances

❌ No Orphan Instances

The Hash Table Problem

```rust
pub trait Hash {
    fn hash<H>(&self, state: &mut H)
    where H: Hasher;
}
```

```rust
impl<T: Display> Hash for T { ... }
```

```rust
impl Hash for u32 { ... }
```

```
error[E0119]: conflicting implementations of trait `Hash` for type `u32`
```

The Hash Table Problem

```rust
pub trait Hash {
    fn hash<H>(&self, state: &mut H)
    where H: Hasher;
}
```

```rust
impl<K, V> HashMap<K, V> {
    pub fn get(&self, k: &K) -> Option<&V>
    where
        K: Hash + Eq,
    { ... }
}
```

```rust
impl Hash for u32 { ... }
```

```rust
impl<T: Display> Hash for T { ... }
```

```rust
fn get_first_value(
    map: HashMap<u32, String>,
    keys: &[u32],
) -> Option<String>
{
    for key in keys {
        if let Some(value) = map.get(key) {
            return Some(value.clone())
        }
    }
    None
}
```

Which instance to use?

Generic Lookup

```rust
impl<T: Display> Hash for T { ... }
```

Instantiation happens here

```rust
fn get_first_value<T: Display + Eq>(
    map: HashMap<T, String>,
    keys: &[T],
) -> Option<String>
{
    for key in keys {
        if let Some(value) = map.get(key) {
            return Some(value.clone())
        }
    }
    None
}
```

T = u32

Instance is forgotten

```rust
impl Hash for u32 { ... }
```

```rust
fn get_first_value(
    map: HashMap<u32, String>,
    keys: &[u32],
) -> Option<String>
{
    for key in keys {
        if let Some(value) = map.get(key) {
            return Some(value.clone())
        }
    }
    None
}
```

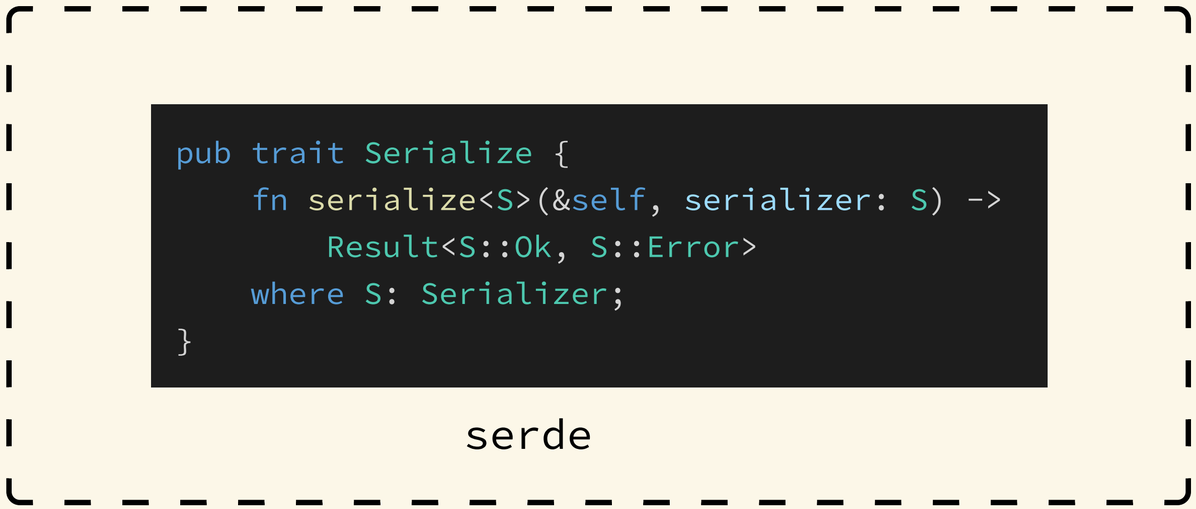
Lookup can be arbitrarily deep

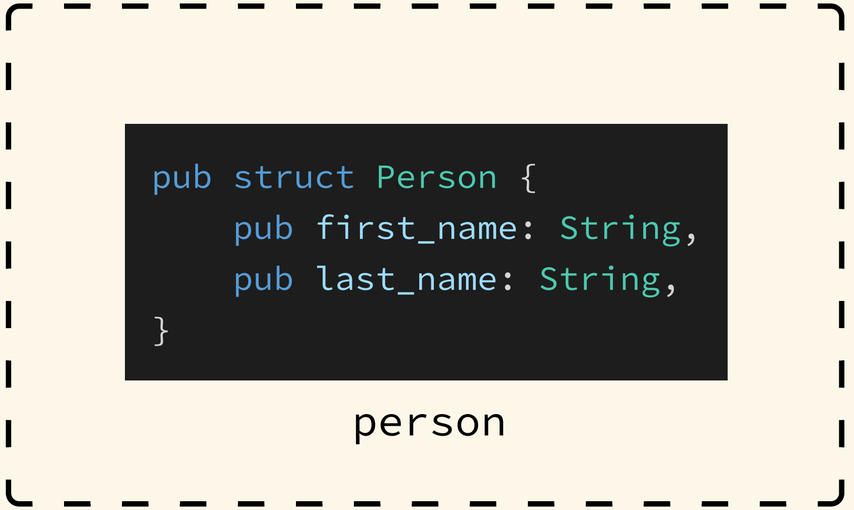
```
fn print_first_value<T: Display + Eq>(
    map: HashMap<T, String>,
    keys: &[T],
) {
    if let Some(value) = get_first_value(map, keys) {
        println!("got first value: {value}")
    }
}
```

The fact that `get_first_value` uses Hash becomes totally obscured

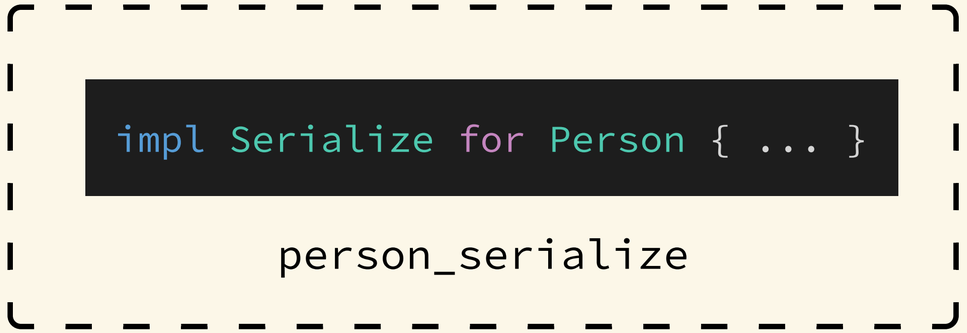
Only way to support this is to perform analysis on the fully monomorphized instances, which may be too expensive

Orphan Rules

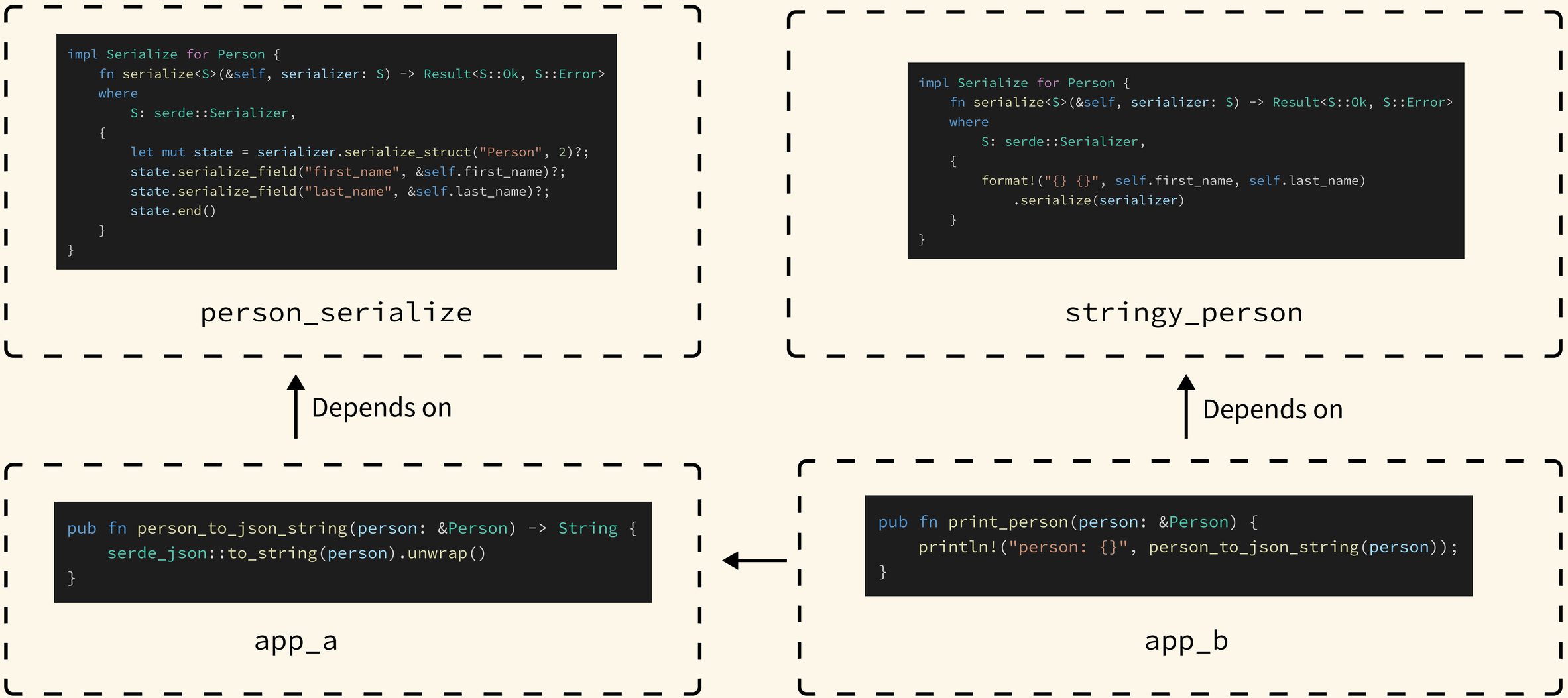




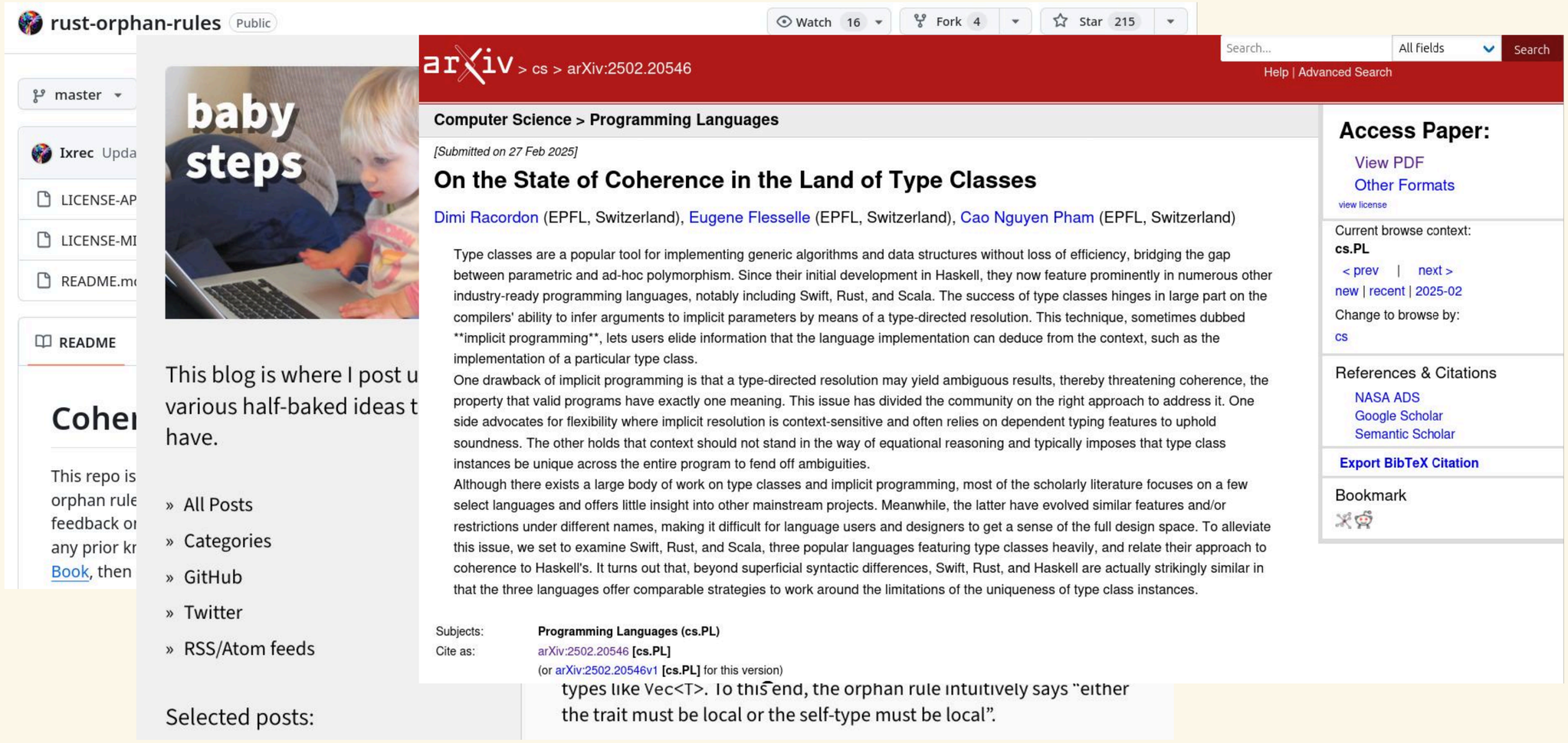
error[E0117]: only traits defined in the current crate can be implemented for types defined outside of the crate



Which Implementation To Choose?



Is Coherence Really a Problem?



https://github.com/Ixrec/rust-orphan-rules https://smallcultfollowing.com/babysteps/blog/2015/01/14/little-orphan-impls/ https://arxiv.org/abs/2502.20546

Overlapping Blanket Implementations can simplify code

```rust
impl<T: AsRef<[u8]>> Serialize for T
{
    fn serialize<S>(&self, serializer: S) -> Result<S::Ok, S::Error>
    where
        S: serde::Serializer,
    {
        serializer.serialize_bytes(self.as_ref())
    }
}
```

```rust
impl<T> Serialize for T
where
    for<'a> &'a T: IntoIterator<Item: Serialize>,
{
    fn serialize<S>(&self, serializer: S) -> Result<S::Ok, S::Error>
    where
        S: Serializer,
```

```rust
ect_seq(self)
```

```rust
impl<T: Display> Serialize for T
{
    fn serialize<S>(&self, serializer: S) -> Result<S::Ok, S::Error>
    where
        S: serde::Serializer,
    {
        self.to_string().serialize(serializer)
    }
}
```

Getting Around Coherence

1. Specialization
2. Explicit Parameters
3. Implicit Parameters
4. Scoped Coherence

Specialization

- Feature Name: specialization
- Start Date: 2015-06-17
- RFC PR: rust-lang/rfcs#1210
- Rust Issue: rust-lang/rust#31844

Summary

This RFC proposes a design for *specialization*, which permits multiple `impl` blocks to apply to the same type/trait, so long as one of the blocks is clearly "more specific" than the other. The more specific `impl` block is used in a case of overlap. The design proposed here also supports refining default trait implementations based on specifics about the types involved.

Altogether, this relatively small extension to the trait system yields benefits for performance and code reuse, and it lays the groundwork for an "efficient inheritance" scheme that is largely based on the trait system (described in a forthcoming companion RFC).

Motivation

Specialization brings benefits along several different axes:

- **Performance**: specialization expands the scope of "zero cost abstraction", because specialized impls can provide custom high-performance code for particular, concrete cases of an abstraction.
- **Reuse**: the design proposed here also supports refining default (but incomplete) implementations of a trait, given details about the types involved.
- **Groundwork**: the design lays the groundwork for supporting "efficient inheritance" through the trait system.

The following subsections dive into each of these motivations in more detail.

https://github.com/rust-lang/rfcs/blob/master/text/1210-impl-specialization.md

#![feature(specialization)]

```rust
pub trait Hash {
    fn hash<H>(&self, state: &mut H)
    where H: Hasher;
}
```

```rust
default impl<T: Display> Hash for T { ... }
```

```rust
impl Hash for u32 { ... }
```

A clear hierarchy of overridable generic implementations

Default ≠ Blanket Implementations

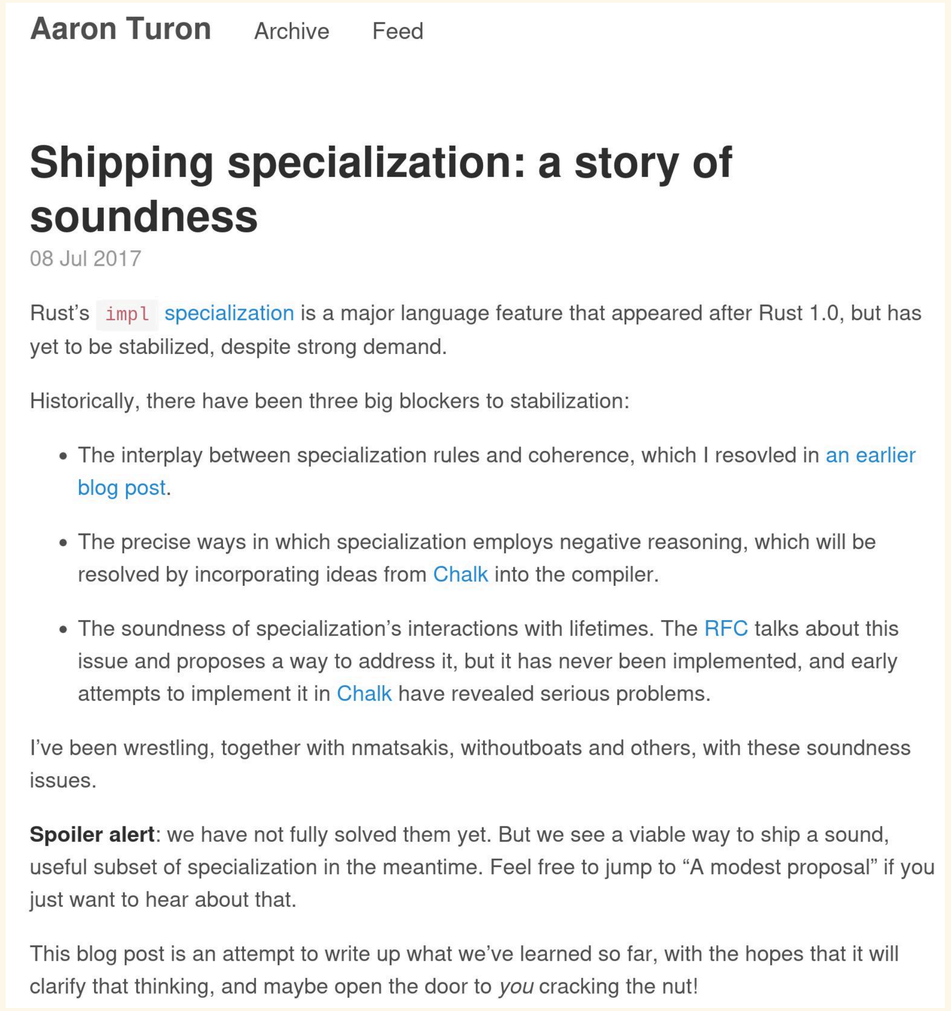
```
default impl<T: Display> Hash for T { ... }
```

```
impl Hash for u32 { ... }
```

```
fn get_first_value<T: Display + Eq>(
    map: HashMap<T, String>,
    keys: &[T],
) -> Option<String>
{
    for key in keys {
        if let Some(value) = map.get(key) {
            return Some(value.clone())
        }
    }
    None
}
```

```
error[E0277]: the trait bound `T: Hash` is not satisfied
```

Specialization Blockers



Aaron Turon Archive Feed

Shipping specialization: a story of soundness

08 Jul 2017

Rust's `impl` specialization is a major language feature that appeared after Rust 1.0, but has yet to be stabilized, despite strong demand.

Historically, there have been three big blockers to stabilization:

- The interplay between specialization rules and coherence, which I resovled in an earlier blog post.
- The precise ways in which specialization employs negative reasoning, which will be resolved by incorporating ideas from Chalk into the compiler.
- The soundness of specialization's interactions with lifetimes. The RFC talks about this issue and proposes a way to address it, but it has never been implemented, and early attempts to implement it in Chalk have revealed serious problems.

I've been wrestling, together with nmatsakis, withoutboats and others, with these soundness issues.

Spoiler alert: we have not fully solved them yet. But we see a viable way to ship a sound, useful subset of specialization in the meantime. Feel free to jump to "A modest proposal" if you just want to hear about that.

This blog post is an attempt to write up what we've learned so far, with the hopes that it will clarify that thinking, and maybe open the door to *you* cracking the nut!

https://aturon.github.io/blog/2017/07/08/lifetime-dispatch/

TLDR

Specialization may be unsound when types or trait bounds contain explicit `'static` lifetimes

`#![feature(min_specialization)]`
may be too restrictive or require unsafe markers

Limitations of Specialization

```rust
default impl<T: Display> Hash for T { ... }
```

```rust
default impl<T> Hash for T
where
    for<'a> &'a T: IntoIterator<Item: Hash>,
{ ... }
```

```
error[E0119]: conflicting
implementations of trait `Hash`
```

Cannot have two equally general default implementations

```rust
// crate_a
pub struct Foo { ... }
```

```rust
// crate_b
impl Hash for Foo { ... }
```

```
error[E0117]: only traits defined in the
current crate can be implemented for
types defined outside of the crate
```

Specialization ≠ No Orphan Rules

Explicit Parameters

- Specialization enables limited forms of overlapping instances
- Orphan instances is not solved with specialization
- Alternative: pass implementations explicitly
- Use case: Serde Remote

Serde Remote

```rust
// crate_a
pub struct Duration {
    pub secs: i64,
    pub nanos: i32,
}
```

```rust
// crate_b
#[derive(Serialize, Deserialize)]
#[serde(remote = "Duration")]
pub struct DurationDef {
    secs: i64,
    nanos: i32,
}
```

```rust
// crate_c
#[derive(Serialize, Deserialize)]
pub struct Process {
    command_line: String,

    #[serde(with = "DurationDef")]
    wall_time: Duration,
}
```

https://serde.rs/remote-derive.html

Serde Remote

```rust
// crate_b
#[derive(Serialize)]
#[serde(remote = "Duration")]
pub struct DurationDef {
    secs: i64,
    nanos: i32,
}
```

```rust
impl DurationDef {
    fn serialize<S>(
        value: &Duration,
        serializer: S,
    ) -> Result<S::Ok, S::Error>
    where
        S: serde::Serializer,
    { ... }
}
```

```rust
#[derive(Serialize)]
struct Process {
    command_line: String,

    #[serde(with = "DurationDef")]
    wall_time: Duration,
}
```

```rust
impl Serialize for Process {
    fn serialize<S>(
        &self,
        serializer: S,
    ) -> Result<S::Ok, S::Error>
    where
        S: Serializer,
    {
        ...
        DurationDef::serialize(self.wall_time, serializer)
        ...
    }
}
```

Some issues with Serde Remote

Pros

- Supports overlapping / orphan implementations

Cons

- `#[serde(remote)]` and `#[serde(with)]` patterns only work with Serde traits
- Other traits require similar heavyweight proc macro machinery
- Weak type-checking when using `#[serde(with)]`
- Inner implementations must be chosen up front

Let’s try to make it easier to support explicit implementations

Provider Traits

```rust
pub trait Serialize {
    fn serialize<S>(
        &self,
        serializer: S,
    ) -> Result<S::Ok, S::Error>
    where
        S: Serializer;
}
```

Consumer Trait

Self becomes Explicit

```rust
pub trait SerializeImpl<T> {
    fn serialize<S>(
        value: &T,
        serializer: S,
    ) -> Result<S::Ok, S::Error>
    where
        S: serde::Serializer;
}
```

Provider Trait

Provider Implementations

```
impl DurationDef {
    fn serialize<S>(
        context: &Duration,
        serializer: S,
    ) -> Result<S::Ok, S::Error>
    where
        S: serde::Serializer,
    { ... }
}
```

Ad Hoc Provider


```
impl SerializeImpl<Duration>
    for DurationDef
{
    fn serialize<S>(
        context: &Duration,
        serializer: S,
    ) -> Result<S::Ok, S::Error>
    where
        S: serde::Serializer
    { ... }
}
```

Named Provider

Well-Typed Provider

Overlapping & Orphan Implementations with Provider Traits

```rust
impl<T: AsRef<[u8]>> Serialize for T
{
    fn serialize<S>(
        &self,
        serializer: S
    ) -> Result<S::Ok, S::Error>
    where
        S: serde::Serializer,
    {
        serializer.serialize_bytes(
            self.as_ref())
    }
}
```

```rust
pub struct SerializeBytes;

impl<T: AsRef<[u8]>> SerializeImpl<T>
    for SerializeBytes
{
    fn serialize<S>(
        value: &T,
        serializer: S
    ) -> Result<S::Ok, S::Error>
    where
        S: serde::Serializer,
    {
        serializer.serialize_bytes(
            value.as_ref())
    }
}
```

T is now a generic parameter

No coherence restrictions when we own Self

Overlapping & Orphan Implementations with Provider Traits

```rust
impl<T> Serialize for T
where
    for<'a> &'a T:
        IntoIterator<Item: Serialize>,
{
    fn serialize<S>(
        &self,
        serializer: S
    ) -> Result<S::Ok, S::Error>
    where
        S: Serializer,
    { ... }
}
```

```rust
pub struct SerializeIterator;

impl<T> SerializeImpl<T>
    for SerializeIterator
where
    for<'a> &'a T:
        IntoIterator<Item: Serialize>,
{
    fn serialize<S>(
        value: &T,
        serializer: S
    ) -> Result<S::Ok, S::Error>
    where
        S: Serializer,
    { ... }
}
```

Problem: Item serialization is still implicit

Higher Order Providers

```
pub struct SerializeIterator<SerializeItem>(
    pub PhantomData<SerializeItem>);

impl<T, SerializeItem> SerializeImpl<T>
    for SerializeIterator<SerializeItem>
where
    for<'a> &'a T: IntoIterator,
    SerializeItem: for<'a>
        SerializeImpl<<&'a T as IntoIterator>::Item>,
{
    fn serialize<S>(
        value: &T,
        serializer: S,
    ) -> Result<S::Ok, S::Error>
    where
        S: Serializer,
    { ... }
}
```

Item serializer is passed explicitly

Implicit Parameters

✓ Explicit parameters enables fully overlapping instances and orphan instances

❌ Tedious to compose with inner implementations

❌ Tedious to pass around

👀 Alternative: find ways to pass implementations *implicitly*

Context & Capabilities

Contexts and capabilities in Rust

Dec 21, 2021

I recently worked out a promising idea with Niko Matsakis and Yoshua Wuyts to solve what I'll call the "context problem" in Rust. The idea takes inspiration from features in other languages like implicit arguments, effects, and object capabilities. While this is very much at the early stages of development, I'm sharing it here to hopefully get more perspectives and ideas from the Rust community.

The problem

Very often in programming we find ourselves needing access to some **context object** in many places in our code. Some examples of context objects:

- Arena allocator
- String interner
- Logging scope
- Async executor
- Capability object

Today we have two main approaches to using these context objects.

https://tmandry.gitlab.io/blog/posts/2021-12-21-context-capabilities/

Context & Capabilities

```
struct BasicArena { .. }

capability basic_arena<'a> = &'a BasicArena;
```

```
with basic_arena = &BasicArena::new() {
    let foo: &Foo = Foo::deserialize(deserializer)?;
}
```

New with keyword

```
impl<'a> Deserialize<'a> for &'a Foo
with
    basic_arena: &'a BasicArena,
{
    fn deserialize<D>(
        deserializer: D,
    ) -> Result<Self, D::Error>
       where D: Deserializer<'de>
    {
        let foo = Foo::from_bytes(
            deserializer.get_bytes())?;
        basic_arena.alloc(foo)
    }
}
```

Parameter passed implicitly

Providers as Capabilities

```
pub struct SerializeIterator;

impl<Context, T> SerializeImpl<T>
    for SerializeIterator
with
    context: &Context,
where
    for<'a> &'a T: IntoIterator,
    Context: for<'a>
        SerializeImpl<<&'a T as IntoIterator>::Item>,
{
    fn serialize<S>(
        value: &T,
        serializer: S,
    ) -> Result<S::Ok, S::Error>
    where
        S: Serializer,
    { ... }
}
```

Only need the provider type

Explicit Context Params

```rust
impl<'a> Deserialize<'a>
    for &'a Foo
with
    basic_arena: &'a BasicArena,
{
    fn deserialize<D>(
        deserializer: D,
    ) -> Result<Self, D::Error>
    where
        D: Deserializer<'de>
    {
        let foo = Foo::from_bytes(
            deserializer.get_bytes())?;
        basic_arena.alloc(foo)
    }
}
```

```rust
pub struct DeserializeFooWithArena;

impl<Context, 'a>
    DeserializeImpl<'a, Context, &'a Foo>
    for DeserializeFooWithArena
where
    Context: HasBasicArena,
{
    fn deserialize<D>(
        context: &Context,
        deserializer: D,
    ) -> Result<&'a Foo, D::Error>
    where
        D: Deserializer<'de>,
    {
        let foo = Foo::from_bytes(
            deserializer.get_bytes())?;
        context.basic_arena().alloc(foo)
    }
}
```

Dependency Injection

Pass context explicitly

Explicit Context Params

```rust
pub struct SerializeIterator;

impl<Context, T> SerializeImpl<Context, T>
    for SerializeIterator
where
    for<'a> &'a T: IntoIterator,
    Context: for<'a>
        SerializeImpl<Context, <&'a T as IntoIterator>::Item>,
{
    fn serialize<S>(
        context: &Context,
        value: &T,
        serializer: S,
    ) -> Result<S::Ok, S::Error>
    where
        S: Serializer,
    { ... }
}
```

Add Context Param

Dependency Injection

Context Providing Implicit Bindings

```
pub struct MyContext {
    pub basic_arena: BasicArena,
    pub logger: Logger,
    ...
}

/*
    Perform the following bindings
    for MyContext:

    SerializeImpl<Vec<Vec<u8>>> -> SerializeIterator
    SerializeImpl<Vec<u8>> -> SerializeBytes
    DeserializeImpl<Foo> -> DeserializeFooWithArena
    ... etc
 */
???
```

```
fn main() {
    let context = MyContext { ... };
    let values: Vec<Vec<u8>> = vec![ ... ];

    // Serialize impl is customized via Context
    let serialized = context.serialize(&values)
        .unwrap();
}
```

Incoherence x Coherence

Key Insight

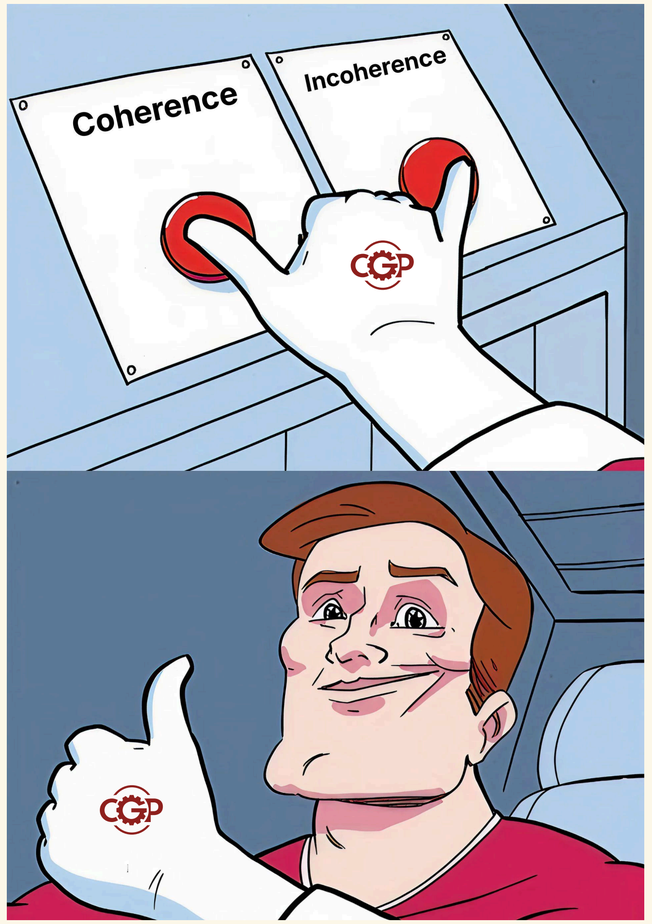
We only want incoherence when **writing** implementations

But we want **scoped** coherence when **using** traits

Solution

✓ Use a provider trait to enable incoherent implementations

👀 Use an explicit context type to provide scoped coherence of dependencies



Introducing Context-Generic Programming (CGP)

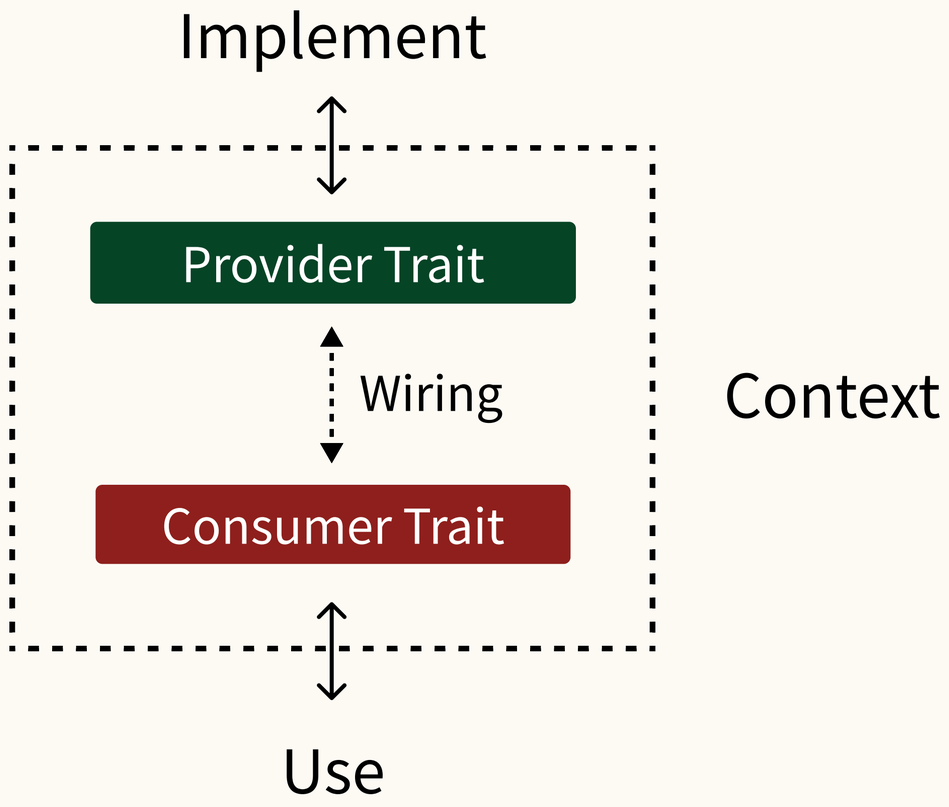
A modular programming paradigm for writing **context-generic** implementations without coherence restrictions

A whole new world of possibilities of how to write modular Rust programs

Key Ideas

1. Introduce Provider Traits for overlapping implementations with unique provider types

2. Additional wiring steps for wiring provider implementations to a concrete context



The `cgp-serde` Crate

- Demonstrates how Serde traits could have been redesigned using CGP.
- Fully backward compatible with original Serde

Used for exploring CGP concepts, **not** for replacing Serde

- The use of Serde may be too widespread for any redesign to be feasible
- There are better ways to design generalized encoding libraries with CGP

The #[cgp_component] Macro

Component Name

```rust
#[cgp_component(ValueSerializer)]
pub trait CanSerializeValue<Value> {
    fn serialize<S>(
        &self,
        value: &Value,
        serializer: S,
    ) -> Result<S::Ok, S::Error>
    where
        S: serde::Serializer;
}
```

Consumer Trait

generates

```rust
pub struct ValueSerializerComponent;
```

generates

```rust
pub trait ValueSerializer<Context, Value> {
    fn serialize<S>(
        context: &Context,
        value: &Value,
        serializer: S,
    ) -> Result<S::Ok, S::Error>
    where
        S: serde::Serializer;
}
```

Provider Trait

Overlapping CGP impls

```
#[cgp_impl(SerializeIterator)]
impl<Context, Value> ValueSerializer<Value>
    for Context
where
    for<'a> &'a Value: IntoIterator,
    Context: for<'a> CanSerializeValue<
        <&'a Value as IntoIterator>::Item>,
{
    fn serialize<S>(
        &self,
        value: &Value,
        serializer: S,
    ) -> Result<S::Ok, S::Error>
    where
        S: serde::Serializer,
    { ... }
}
```

Overlapping Impl with provider trait and `#[cgp_impl]`

Dependency injection of the consumer trait via Context

Desugaring Provider Impls

```rust
#[cgp_impl(SerializeIterator)]
impl<Context, Value> ValueSerializer<Value>
    for Context
where
    for<'a> &'a Value: IntoIterator,
    Context: for<'a> CanSerializeValue<
        <&'a Value as IntoIterator>::Item>,
{
    fn serialize<S>(
        &self,
        value: &Value,
        serializer: S,
    ) -> Result<S::Ok, S::Error>
    where
        S: serde::Serializer,
    { ... }
}
```

Desugars to

```rust
impl<Context, Value>
    ValueSerializer<Context, Value>
    for SerializeIterator
where
    for<'a> &'a Value: IntoIterator,
    Context: for<'a> CanSerializeValue<
        <&'a Value as IntoIterator>::Item>,
{
    fn serialize<S>(
        context: &Context,
        value: &Value,
        serializer: S,
    ) -> Result<S::Ok, S::Error>
    where
        S: serde::Serializer,
    { ... }
}
```

CGP Contexts

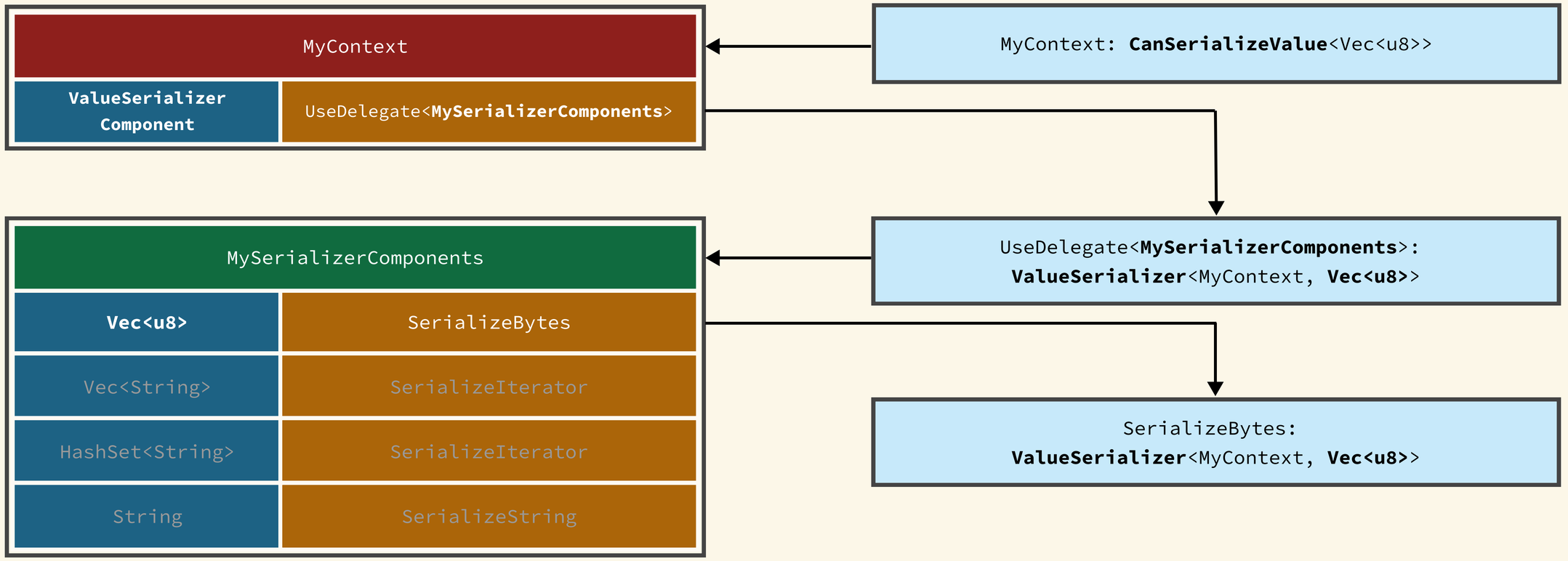
```
pub struct MyContext { ... }

delegate_components! {
  MyContext {
    ValueSerializerComponent:
      UseDelegate<
        new MySerializerComponents {
          Vec<u8>:
            SerializeBytes,
          [
            Vec<String>,
            HashSet<String>,
          ]:
            SerializeIterator,
          String:
            SerializeStr,
        }> } }
```

Type-Level Lookup Tables

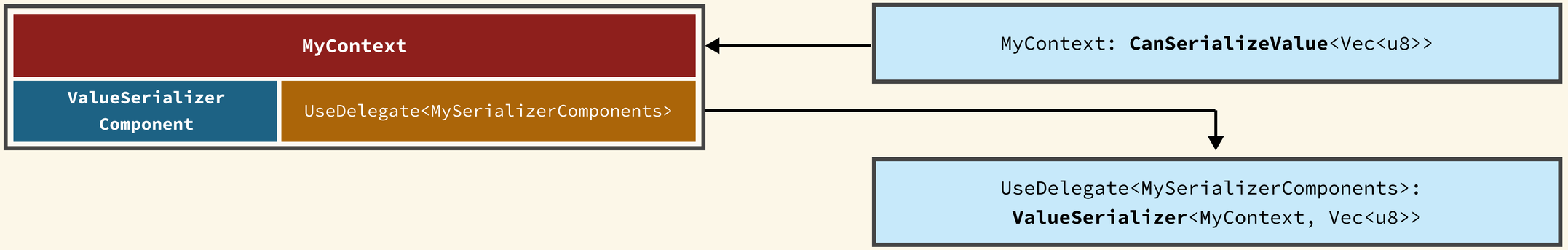
MyContext	
ValueSerializer Component	UseDelegate<SerializerComponents>

MySerializerComponents	
Vec<u8>	SerializeBytes
Vec<String>	SerializeIterator
HashSet<String>	SerializeIterator
String	SerializeString

Type-Level Lookup Tables



Consumer Trait Lookup

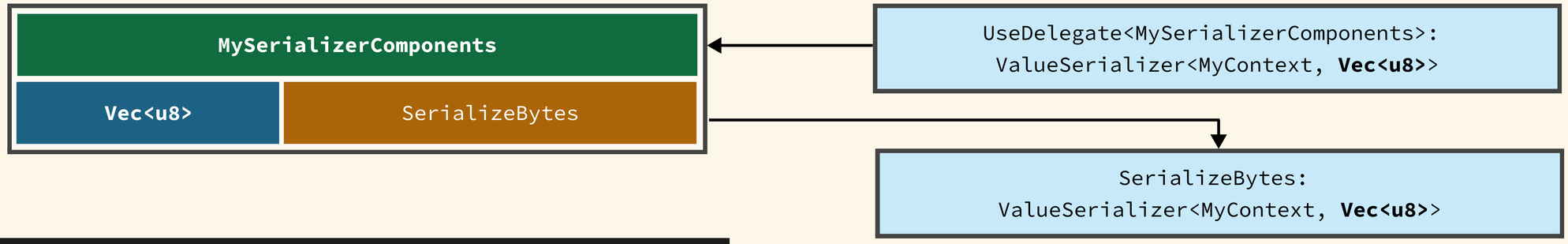



```rust
impl<Context, Value> CanSerializeValue<Value> for Context
where
    Context: DelegateComponent<ValueSerializerComponent>,
    Context::Delegate: ValueSerializer<Context, Value>,
{
    fn serialize<S>(
        &self,
        value: &Value, serializer: S,
    ) -> Result<S::Ok, S::Error>
    where
        S: serde::Serializer,
    {
        Context::Delegate::serialize(self, value, serializer)
    }
}
```

```rust
pub trait DelegateComponent<Name> {
    type Delegate;
}
```

```rust
impl DelegateComponent<ValueSerializerComponent>
    for MyContext
{
    type Delegate =
        UseDelegate<MySerializerComponents>;
}
```

UseDelegate Lookup




```rust
#[cgp_impl(UseDelegate<Components>)]
impl<Context, Value, Components>
    ValueSerializer<Value> for Context
where
    Components: DelegateComponent<Value>,
    Components::Delegate: ValueSerializer<Context, Value>,
{
    fn serialize<S>(
        &self,
        value: &Value, serializer: S,
    ) -> Result<S::Ok, S::Error>
    where
        S: serde::Serializer,
    {
        Components::Delegate::serialize(
            self, value, serializer)
    }
}
```

```rust
pub struct MySerializerComponents;

impl DelegateComponent<Vec<u8>>
    for MySerializerComponents
{
    type Delegate = SerializeBytes;
}
```

Demo

Modular Serialization with `cgp-serde`

Let's build a naive encrypted messaging library

```rust
pub struct EncryptedMessage {
    pub message_id: u64,
    pub author_id: u64,
    pub date: DateTime<Utc>,
    pub encrypted_data: Vec<u8>,
}
```

Deeply Nested Field Customizations

```rust
pub struct MessagesByTopic {
    pub encrypted_topic: Vec<u8>,
    pub messages: Vec<EncryptedMessage>,
}
```

```rust
pub struct MessagesArchive {
    pub decryption_key: Vec<u8>,
    pub messages_by_topics: Vec<MessagesByTopic>,
}
```

All bytes should be serialized the same way

Feature Request: Library users want different encodings


```
{
  "decryption_key": "746f702d736563726574",
  "messages_by_topics": [
    {
      "encrypted_topic": "416c6c2061626f757420434750",
      "messages": [
        {
          "message_id": 1,
          "author_id": 2,
          "date": "2025-11-03T14:15:00+00:00",
          "encrypted_data":
            "48656c6c6f2066726f6d20527573744c616221"
        },
        {
          "message_id": 4,
          "author_id": 8,
          "date": "2025-12-19T23:45:00+00:00",
          "encrypted_data":
            "4f6e652079656617220616e6e697665727361727921"
        }
      ]
    }
  ]
}
```


App A: Hex bytes and RFC3339 Date


```
{
  "decryption_key": "dG9wLXNlY3JldA==",
  "messages_by_topics": [
    {
      "encrypted_topic": "QWxsIGFib3V0IENHUA==",
      "messages": [
        {
          "message_id": 1,
          "author_id": 2,
          "date": 1762179300,
          "encrypted_data": "SGVsbG8gZnJvbSBSdXN0TGFiIQ=="
        },
        {
          "message_id": 4,
          "author_id": 8,
          "date": 1766187900,
          "encrypted_data": "T25lIHllYXIgYW5uaXZlcnNhcnkh"
        }
      ]
    }
  ]
}
```


App B: Base64 bytes and Unix timestamps

Concrete Implementations

```
pub struct AppA;

delegate_components! {
  AppA {
    ValueSerializerComponent:
      UseDelegate<new SerializerComponentsA {
        Vec<u8>: SerializeHex,
        DateTime<Utc>: SerializeRfc3339Date,
        [ u64, String ]: UseSerde,
        <'a, T> &'a T: SerializeDeref,
        [
          Vec<EncryptedMessage>,
          Vec<MessagesByTopic>
        ]:
          SerializeIterator,
        [
          MessagesArchive,
          MessagesByTopic,
          EncryptedMessage
        ]:
          SerializeFields,
      }>
  }
}
```

```
pub struct AppB;

delegate_components! {
  AppB {
    ValueSerializerComponent:
      UseDelegate<new SerializerComponentsB {
        Vec<u8>: SerializeBase64,
        DateTime<Utc>: SerializeTimestamp,
        [ i64, u64, String ]: UseSerde,
        <'a, T> &'a T: SerializeDeref,
        [
          Vec<EncryptedMessage>,
          Vec<MessagesByTopic>,
        ]:
          SerializeIterator,
        [
          MessagesArchive,
          MessagesByTopic,
          EncryptedMessage
        ]:
          SerializeFields,
      }>
  }
}
```

Serializing with Context

```
serde_json::to_string_pretty(
    &SerializeWithContext::new(
        &AppA, &value))
```


```
{
  "decryption_key": "746f702d736563726574",
  "messages_by_topics": [
    {
      "encrypted_topic": "416c6c2061626f757420434750",
      "messages": [
        {
          "message_id": 1,
          "author_id": 2,
          "date": "2025-11-03T14:15:00+00:00",
          "encrypted_data":
            "48656c6c6f2066726f6d20527573744c616221"
        },
        {
          "message_id": 4,
          "author_id": 8,
          "date": "2025-12-19T23:45:00+00:00",
          "encrypted_data":
            "4f6e652079656172206e6e697665727361727921"
        }
      ]
    }
  ]
}
```


```
serde_json::to_string_pretty(
    &SerializeWithContext::new(
        &AppB, &value))
```


```
{
  "decryption_key": "dG9wLXNlY3JldA==",
  "messages_by_topics": [
    {
      "encrypted_topic": "QWxsIGFib3V0IENHUA==",
      "messages": [
        {
          "message_id": 1,
          "author_id": 2,
          "date": 1762179300,
          "encrypted_data": "SGVsbG8gZnJvbSBSdXN0TGFiIQ=="
        },
        {
          "message_id": 4,
          "author_id": 8,
          "date": 1766187900,
          "encrypted_data": "T25lIHllYXIgYW5uaXZlcnNhcnkh"
        }
      ]
    }
  ]
}
```

You don't even need `#[derive(Serialize)]`

```rust
#[derive(CgpData)]
pub struct EncryptedMessage {
    pub message_id: u64,
    pub author_id: u64,
    pub date: DateTime<Utc>,
    pub encrypted_data: Vec<u8>,
}
```

```rust
#[derive(CgpData)]
pub struct MessagesByTopic {
    pub encrypted_topic: Vec<u8>,
    pub messages: Vec<EncryptedMessage>,
}
```

```rust
#[derive(CgpData)]
pub struct MessagesArchive {
    pub decryption_key: Vec<u8>,
    pub messages_by_topics: Vec<MessagesByTopic>,
}
```

Extensible Data Types

One derive works with any CGP trait

Conclusion

CGP makes it easy to work with both coherence and incoherence

```rust
#[cgp_component(HashProvider)]
pub trait Hash { ... }
```

```rust
#[cgp_impl(new HashWithDisplay)]
impl<T: Display> HashProvider
    for T { ... }
```

```rust
impl Hash for u32 { ... }
```

```rust
pub struct MyData { ... }

impl Display for MyData { ... }

delegate_components! {
    MyData {
        HashProviderComponent:
            HashWithDisplay,
    }
}
```

Getting Started with CGP

```
[dependencies]
cgp = "0.6.0"
```

```
use cgp::prelude::*;
```

- Use **`#[cgp_component]`** on any supported trait
- Use **`#[cgp_impl]`** to write named overlapping/ orphan impls
- Use **`delegate_components!`** to use a named impl
- Vanilla trait impls work as usual

Add extra **`Context`** parameter to enable advanced capabilities

New Possibilities with CGP

- Abstract Types
- Modular error handling
- Meta-framework for building other frameworks
  - Web, UI, encoding, games, etc.
- Type-level DSLs
- Extensible records & variants
  - Solves the expression problem
- LLM inference server
- And more..

Challenges of CGP

⚠️ Verbose error messages

LLMs help understanding root causes quickly

😨 Steep learning curve

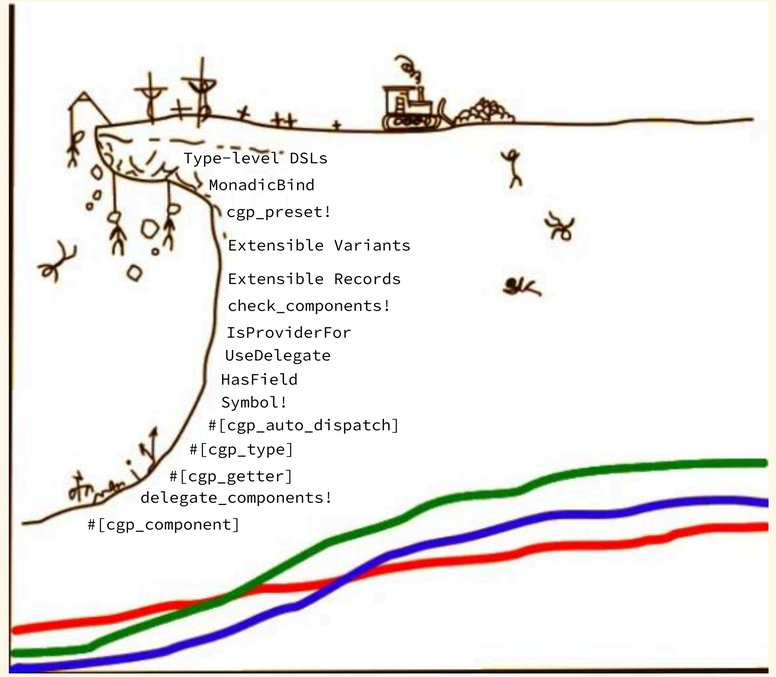
Generics and traits alone are hard enough!

🤯 Almost a different language

Like adding traits or lifetime to a language without it

🚧 Early stage community

Need help spreading knowledge and awareness



Call for contributions!

Related Work

CGP builds on top of many existing programming concepts

Functional Programming

Typeclasses

Implicit Parameters

ML Modules

Type-Level Programming

Tagless Final

Algebraic Effects

Object-Oriented Programming

Inheritance

Mixins

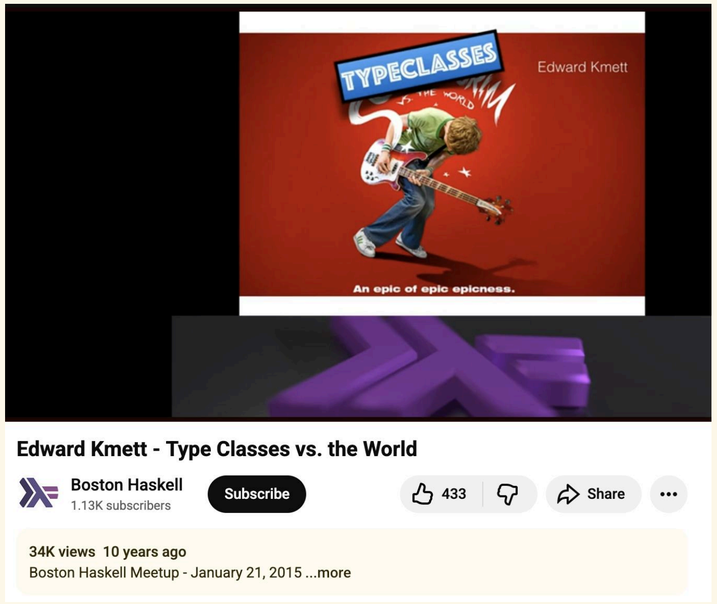
Prototypes

Dependency Injection

Duck Typing

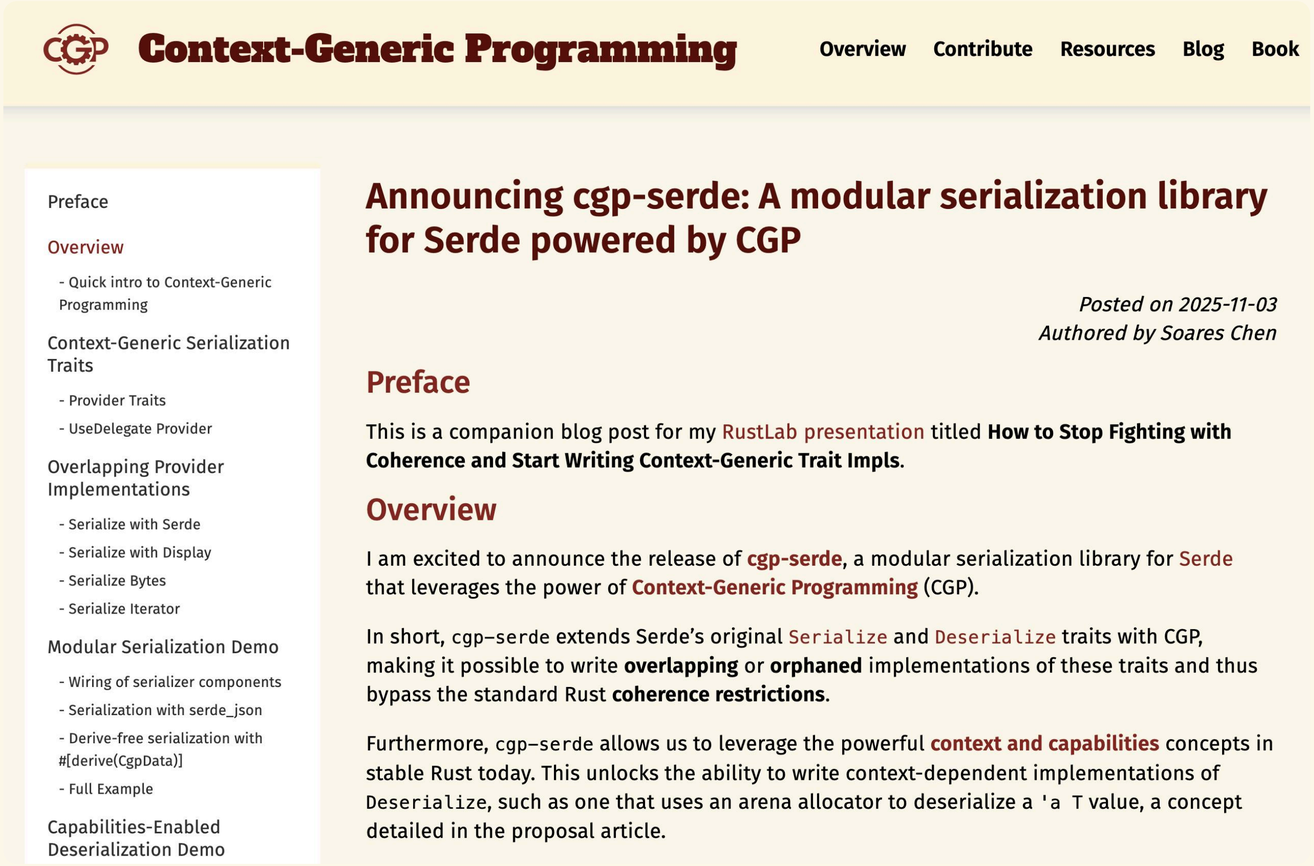
Typeclasses vs the World

by Edward Kmett



https://youtu.be/hIZxTQP1ifo

Releasing `cgp-serde` @ RustLab



Context-Generic Programming

Overview Contribute Resources Blog Book

Preface

Overview
- Quick intro to Context-Generic Programming

Context-Generic Serialization Traits
- Provider Traits
- UseDelegate Provider

Overlapping Provider Implementations
- Serialize with Serde
- Serialize with Display
- Serialize Bytes
- Serialize Iterator

Modular Serialization Demo
- Wiring of serializer components
- Serialization with serde_json
- Derive-free serialization with #[derive(CgpData)]
- Full Example

Capabilities-Enabled Deserialization Demo

Announcing cgp-serde: A modular serialization library for Serde powered by CGP

Posted on 2025-11-03
Authored by Soares Chen

Preface

This is a companion blog post for my RustLab presentation titled **How to Stop Fighting with Coherence and Start Writing Context-Generic Trait Impls**.

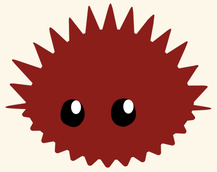
Overview

I am excited to announce the release of **cgp-serde**, a modular serialization library for Serde that leverages the power of **Context-Generic Programming** (CGP).

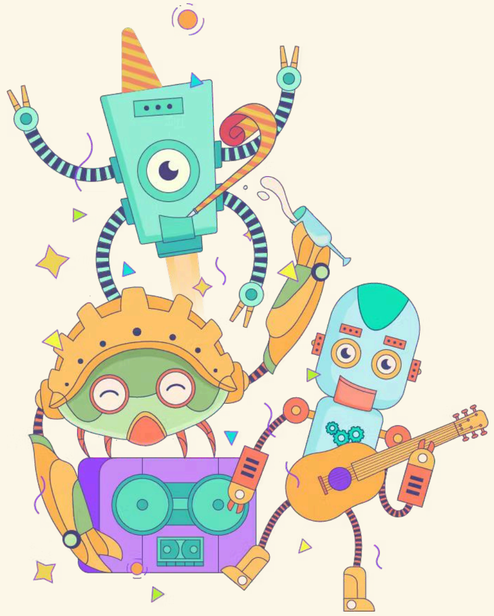
In short, `cgp-serde` extends Serde's original `Serialize` and `Deserialize` traits with CGP, making it possible to write **overlapping** or **orphaned** implementations of these traits and thus bypass the standard Rust **coherence restrictions**.

Furthermore, `cgp-serde` allows us to leverage the powerful **context and capabilities** concepts in stable Rust today. This unlocks the ability to write context-dependent implementations of `Deserialize`, such as one that uses an arena allocator to deserialize a `'a T` value, a concept detailed in the proposal article.

https://contextgeneric.dev/blog/cgp-serde-release/



Thank You for Listening!



Join us at https://contextgeneric.dev