



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 & PLT
- Creator of Context-Generic Programming



Agenda

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

Rust Traits

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

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

```
pub trait Iterator {  
    type Item;  
  
    fn next(&mut self) -> Option<Self::Item>;  
    ...  
}
```

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



What are Traits?

define shared behavior
in an abstract way

Implement



Trait



Use

similar to interfaces,
but with some differences

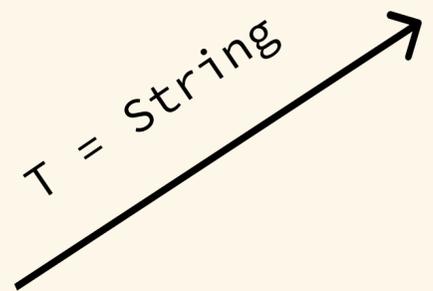
a generic type can be any type
that has certain behavior

The screenshot shows the Rust Programming Language documentation page for "Traits: Defining Shared Behavior". The page title is "Traits: Defining Shared Behavior". The main text explains that a *trait* defines the functionality a particular type has and can share with other types. It also mentions that traits can be used to define shared behavior in an abstract way and that *trait bounds* can be used to specify that a generic type can be any type that has certain behavior. A note states: "Note: Traits are similar to a feature often called *interfaces* in other languages, although with some differences." The page also includes a section titled "Defining a Trait" which explains that a type's behavior consists of the methods we can call on that type. It also provides an example of 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. The page footer mentions: "We want to make a media aggregator library crate named `aggregator` that can display summaries".



Why Generics

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



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



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

Monomorphization



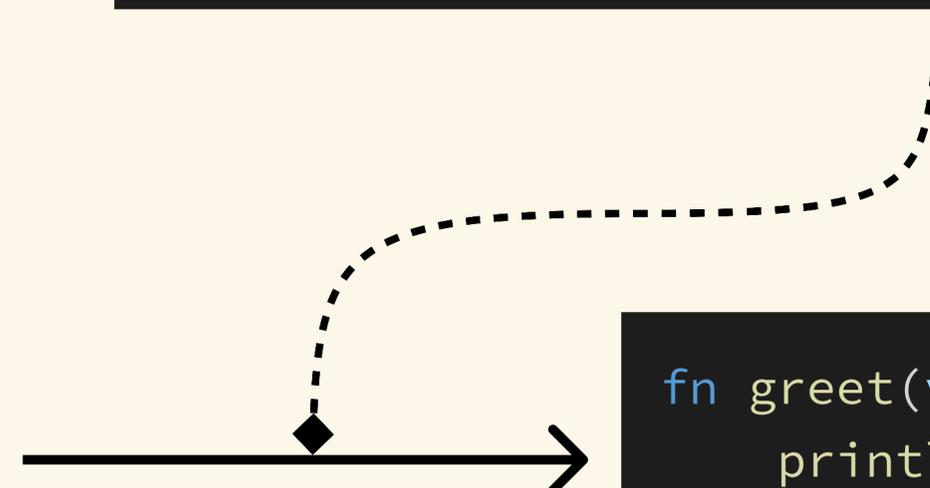
Implementing Traits

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

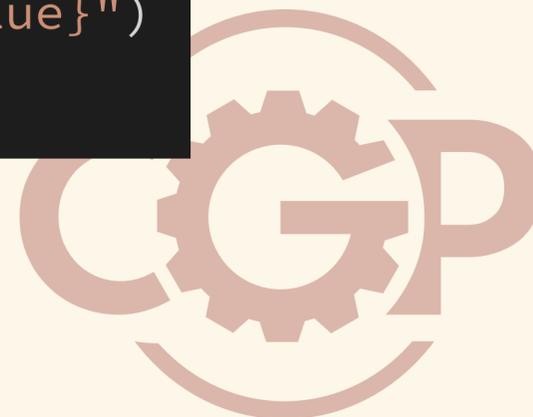
```
impl Display for Person {  
    fn fmt(&self, f: &mut Formatter<'_>) -> Result<(), Error> {  
        write!(f, "{} {}", self.first_name, self.last_name)  
    }  
}
```

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

T = Person



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



Generic Trait Implementations

Uses generics & implement traits generically

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

```
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

```
impl<Name> Display for Person<Name>  
where  
    Name: Display,  
{ ... }
```

Name = String

```
impl Display for String { ... }
```

```
impl Display for Person<String> { ... }
```

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

T = Person<String>

```
fn greet(value: &Person<String>) {  
    println!("Hello {value}")  
}
```



Dependency Injection with Rust Traits

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

No requirement of
Display on Name here

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

No requirement of
Display on Name here

```
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,  
        )  
    }  
}
```

Hidden Dependencies

Transitive Dependencies Lookup

Globally Unique Instance

```
impl Display for String { ... }
```

Transitive Dependencies

```
impl Display for Person<String> { ... }
```

Direct Dependencies

```
fn greet(value: &Person<String>) {  
    println!("Hello {value}")  
}
```

Globally Unique Instance

```
impl<Name> Display for Person<Name>  
where  
    Name: Display,  
{ ... }
```

Name = String

T = Person<String>

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



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

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

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

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

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



The Hash Table Problem

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

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

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

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

```
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

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

Instantiation happens here

```
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

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

Instance is forgotten

```
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

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

serde

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

person

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

```
impl Serialize for Person { ... }
```

person_serialize

Which Implementation To Choose?

```
impl Serialize for Person {  
    fn serialize<S>(&self, serializer: S) -> Result<S::Ok, S::Error>  
    where  
        S: serde::Serializer,  
    {  
        let mut state = serializer.serialize_struct("Person", 2)?;  
        state.serialize_field("first_name", &self.first_name)?;  
        state.serialize_field("last_name", &self.last_name)?;  
        state.end()  
    }  
}
```

person_serialize

↑ Depends on

```
pub fn person_to_json_string(person: &Person) -> String {  
    serde_json::to_string(person).unwrap()  
}
```

app_a

```
impl Serialize for Person {  
    fn serialize<S>(&self, serializer: S) -> Result<S::Ok, S::Error>  
    where  
        S: serde::Serializer,  
    {  
        format!("{}", self.first_name, self.last_name)  
            .serialize(serializer)  
    }  
}
```

stringy_person

↑ Depends on

```
pub fn print_person(person: &Person) {  
    println!("person: {}", person_to_json_string(person));  
}
```

app_b



Is Coherence Really a Problem?

The image shows a composite of two web pages. On the left is a GitHub repository for 'rust-orphan-rules', which is public. It features a sidebar with navigation links for 'LICENSE-A', 'LICENSE-M', 'README.m', and 'README'. The main content area has a header 'baby steps' and a sub-header 'Cohere'. Below this is a list of links: 'All Posts', 'Categories', 'GitHub', 'Twitter', and 'RSS/Atom feeds'. A section titled 'Selected posts:' is partially visible. On the right is an arXiv paper page for 'On the State of Coherence in the Land of Type Classes' by Dimi Racordon, Eugene Flesselle, and Cao Nguyen Pham. The paper is in the 'Computer Science > Programming Languages' category and was submitted on 27 Feb 2025. The abstract discusses type classes and coherence in Haskell, Swift, Rust, and Scala. The paper is available in PDF and other formats, and can be cited via NASA ADS, Google Scholar, or Semantic Scholar. A bookmark section is also present.



This blog is where I post up various half-baked ideas that I have.

- » All Posts
- » Categories
- » GitHub
- » Twitter
- » RSS/Atom feeds

Selected posts:

you can implement `MyTrait` for any type you like, including external types like `Vec<T>`. To this end, the orphan rule intuitively says “either the trait must be local or the self-type must be local”.

Overlapping Blanket Implementations can simplify code

```
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())
    }
}
```

```
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,
```

```
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)
    }
}
```

```
    .collect_seq(self)
```

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)]

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

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

```
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 resolved 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

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

```
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



Specialization ≠ No Orphan Rules

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

```
// 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



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

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

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

```
// 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

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

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

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

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

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

Assume method signature is correct

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 upfront

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



Provider Traits

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

Consumer Trait

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

Self becomes Explicit

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

```
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())
        }
}
```



```
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())
        }
}
```

No coherence restrictions
when we own Self



Overlapping & Orphan Implementations with Provider Traits

```
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,
    { ... }
}
```



```
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

New with keyword

```
struct BasicArena { .. }  
  
capability basic_arena<'a> = &'a BasicArena;
```

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

```
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

```
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)
  }
}
```



```
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

```
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 implementations

Solution

- ✓ Use a provider trait to enable incoherent implementations
- 👁️ Use an explicit context type to provide scoped coherence of dependencies





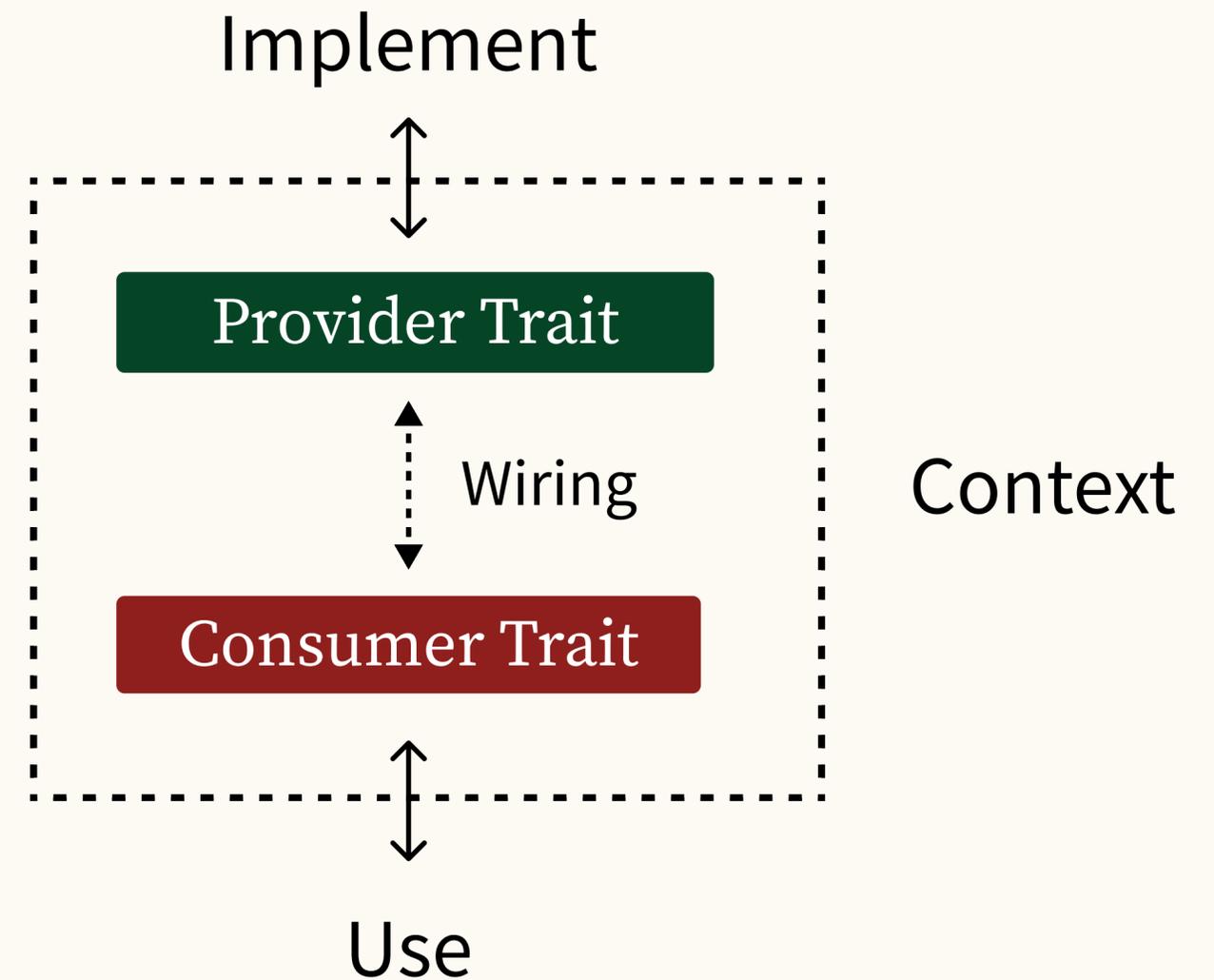
Introducing
Context-Generic
Programming
(CGP)

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



The #[cgp_component] Macro

Component Name

```
#[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

```
pub struct ValueSerializerComponent;
```

generates

```
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

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

```
#[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,
    { ... }
}
```

Dependency injection of the
consumer trait via Context



Desugaring Provider 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,
    { ... }
}
```

Desugars to



```
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 { ... }

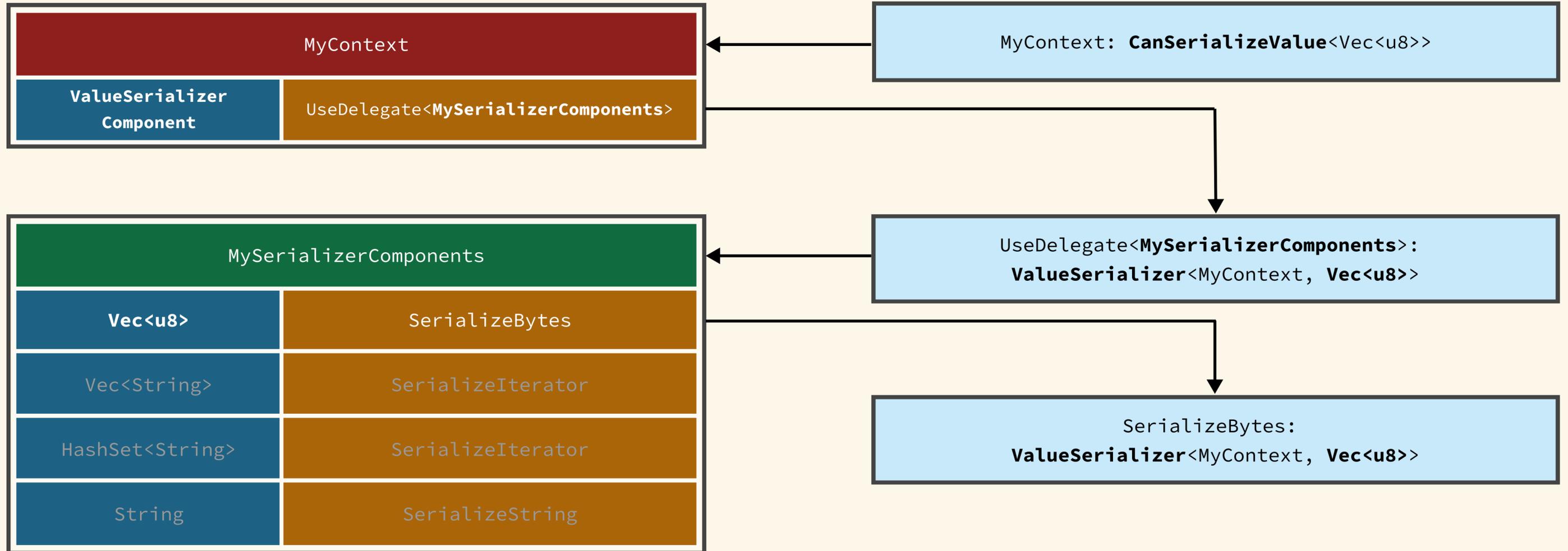
delegate_components! {
  MyContext {
    ValueSerializerComponent:
      UseDelegate<
        new MySerializerComponents {
          Vec<u8>:
            SerializeBytes,
          [
            Vec<String>,
            HashSet<String>,
          ]:
            SerializeIterator,
          String:
  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

Type-Level Lookup Tables

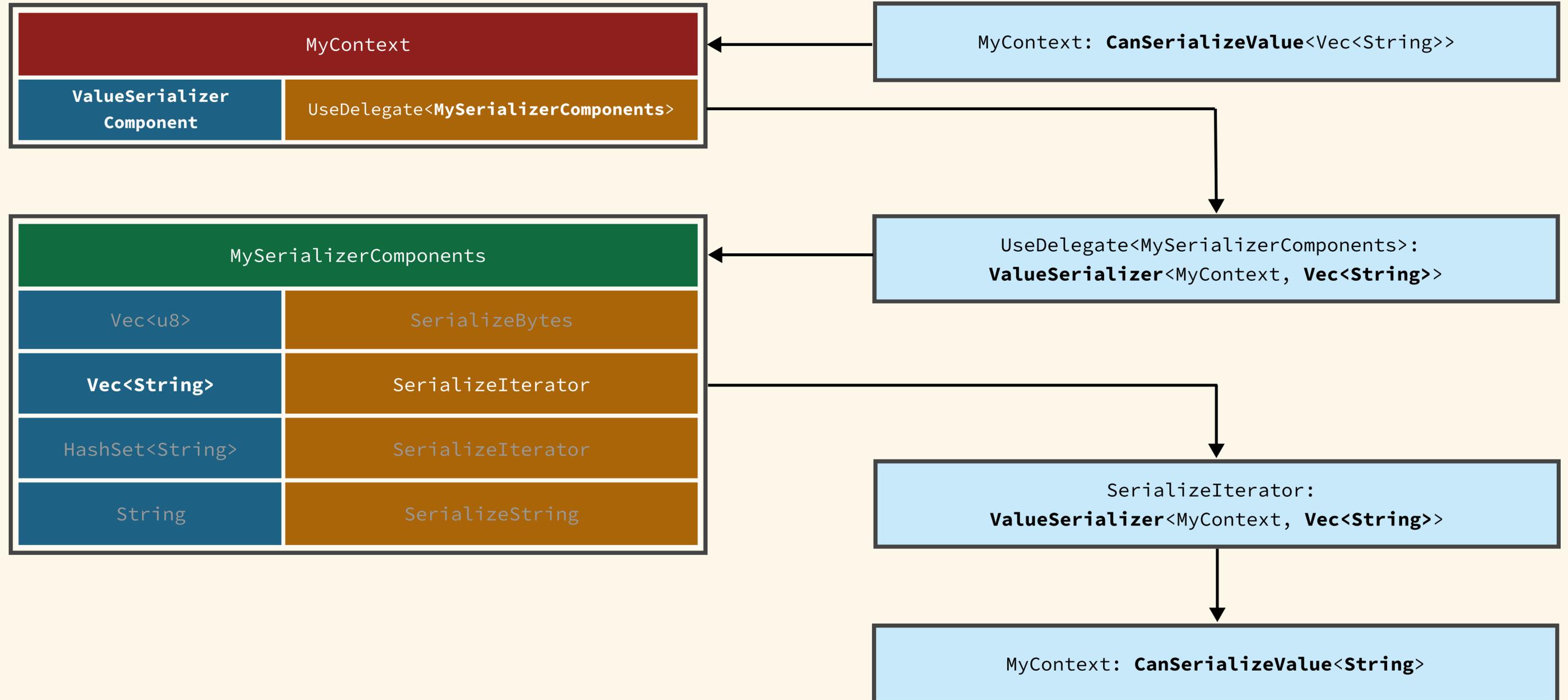
MyContext	
ValueSerializerComponent	UseDelegate<SerializerComponents>

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

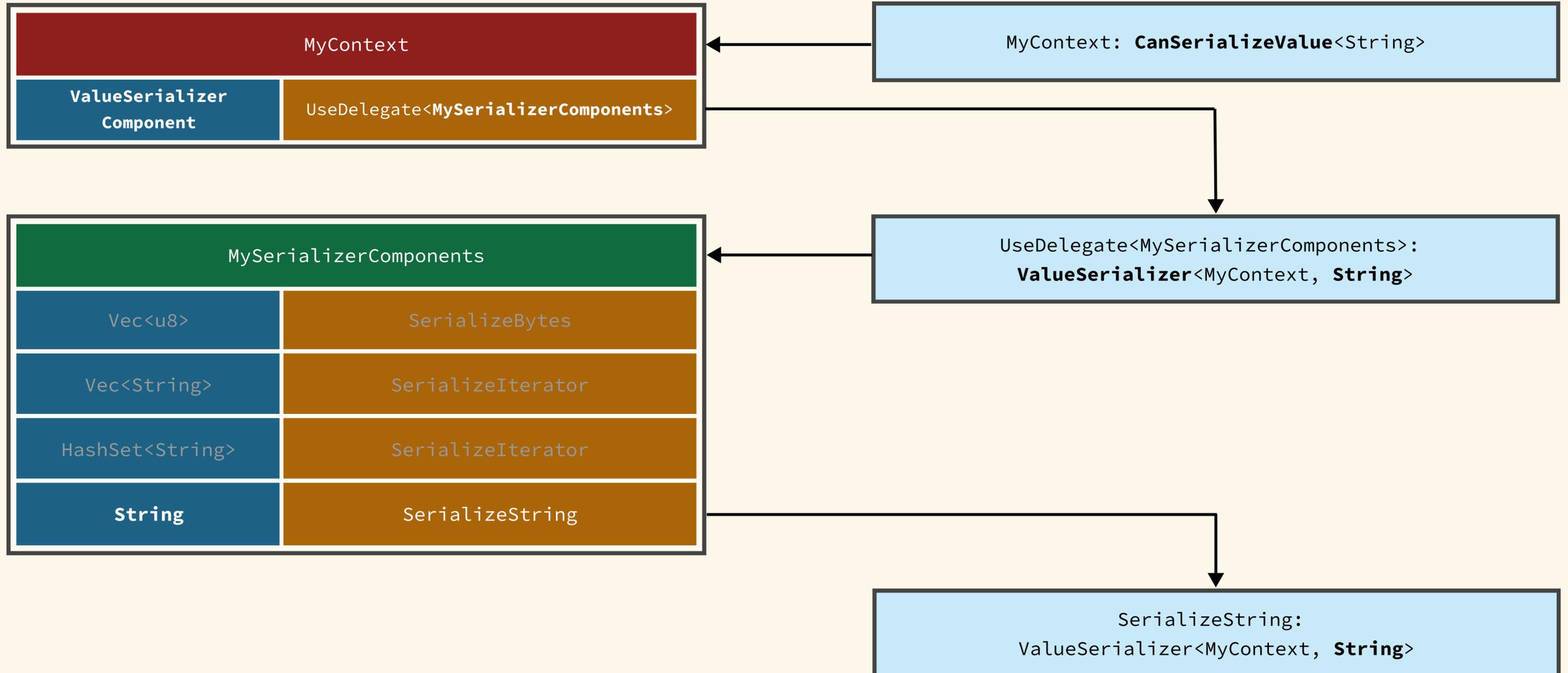
Type-Level Lookup Tables



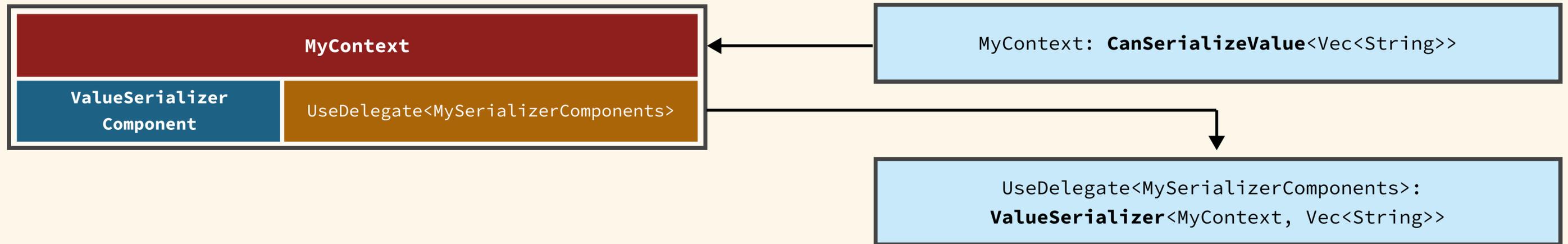
Type-Level Lookup Tables



Type-Level Lookup Tables



Consumer Trait Lookup

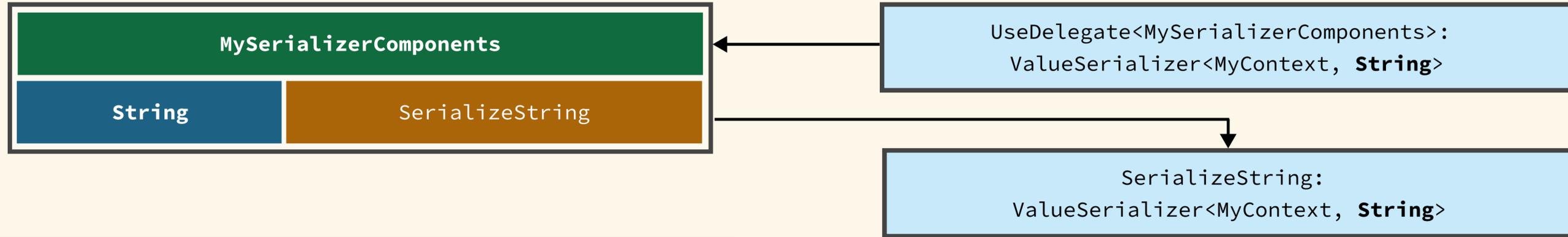


```
impl<Context, Value> CanSerializeValue<Value> for Context
where
  Context: DelegateComponent<ValueSerializerComponent>,
  Context::Delegate: ValueSerializer<Context, Value>,
{
  fn serialize<S>(
    &self,
    value: &Value, serializer: S,
  ) -> Result<S::Ok, S::Error>
  where
    S: serde::Serializer,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

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

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

UseDelegate Lookup



```
#[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)
  }
}
```

```
pub struct MySerializerComponents;

impl DelegateComponent<String>
  for MySerializerComponents
{
  type Delegate = SerializeString;
}
```



Demo

Modular Serialization with `cgp-serde`



Let's build a naive encrypted messaging library

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

Deeply Nested Field
Customizations

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

```
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": "7365637265742d6465616c73",
      "messages": [
        {
          "message_id": 1,
          "author_id": 4,
          "date": "2025-01-01T00:00:00+00:00",
          "encrypted_data": "627579203120667265652031"
        },
        {
          "message_id": 2,
          "author_id": 8,
          "date": "2025-01-01T00:00:00+00:00",
          "encrypted_data": "73616c657320737461727420746f6d6f72726f77"
        }
      ]
    }
  ]
}
```

App A: Hex bytes and RFC3339 Date

```
{
  "decryption_key": "dG9wLXNlY3JldA==",
  "messages_by_topics": [
    {
      "encrypted_topic": "c2VjcmV0LWRlYWxz",
      "messages": [
        {
          "message_id": 1,
          "author_id": 4,
          "date": 1735689600,
          "encrypted_data": "YnV5IDEgZnJlZSAx"
        },
        {
          "message_id": 2,
          "author_id": 8,
          "date": 1735689600,
          "encrypted_data": "c2FsZXMgc3RhcncQgdG9tb3Jyb3c="
        }
      ]
    }
  ]
}
```

App B: Base64 bytes and Unix timestamps

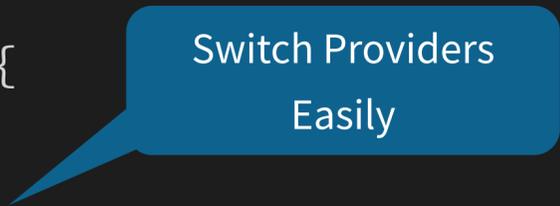
App A Implementation

```
pub struct AppA;

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

App B Implementation

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



Switch Providers
Easily

Serializing with App A

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

```
{  
  "decryption_key": "746f702d736563726574",  
  "messages_by_topics": [  
    {  
      "encrypted_topic": "7365637265742d64656616c73",  
      "messages": [  
        {  
          "message_id": 1,  
          "author_id": 1,  
          "date": "2025-01-01T00:00:00+00:00",  
          "encrypted_data": "627579203120667265652031"  
        },  
        {  
          "message_id": 2,  
          "author_id": 8,  
          "date": "2025-01-01T00:00:00+00:00",  
          "encrypted_data": "73616c6573207374617227420746f6d6f72726f77"  
        }  
      ]  
    }  
  ]  
}
```

Serializing with App B

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

```
{  
  "decryption_key": "dG9wLXNlY3JldA==",  
  "messages_by_topics": [  
    {  
      "encrypted_topic": "c2VjcmV0LWRlYWxz",  
      "messages": [  
        {  
          "message_id": 1,  
          "author_id": 1,  
          "date": 1735689600,  
          "encrypted_data": "YnV5IDEGznJlZSAx"  
        },  
        {  
          "message_id": 2,  
          "author_id": 8,  
          "date": 1735689600,  
          "encrypted_data": "c2FsZXMgc3RhcnQgdG9tb3Jyb3c="        }  
      ]  
    }  
  ]  
}
```

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

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

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

```
#[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

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

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

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

```
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
- Meta-framework for building other frameworks
 - Web, UI, encoding, games, etc.
- Type-level DSLs
- Extensible Records & Variants
- AI Inference & Training
- 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

Similar impact as OOP on C, or Rails on Ruby

🚧 Early stage community

Need help spreading knowledge and awareness

Call for contribution!

⚠️ Use it at
your own risk!



Related Work

CGP builds on top of many existing programming concepts

Typeclasses

Implicit Parameters

ML Modules

Type-Level Programming

Tagless Final

Algebraic Effects

Object-Oriented Programming

Inheritance

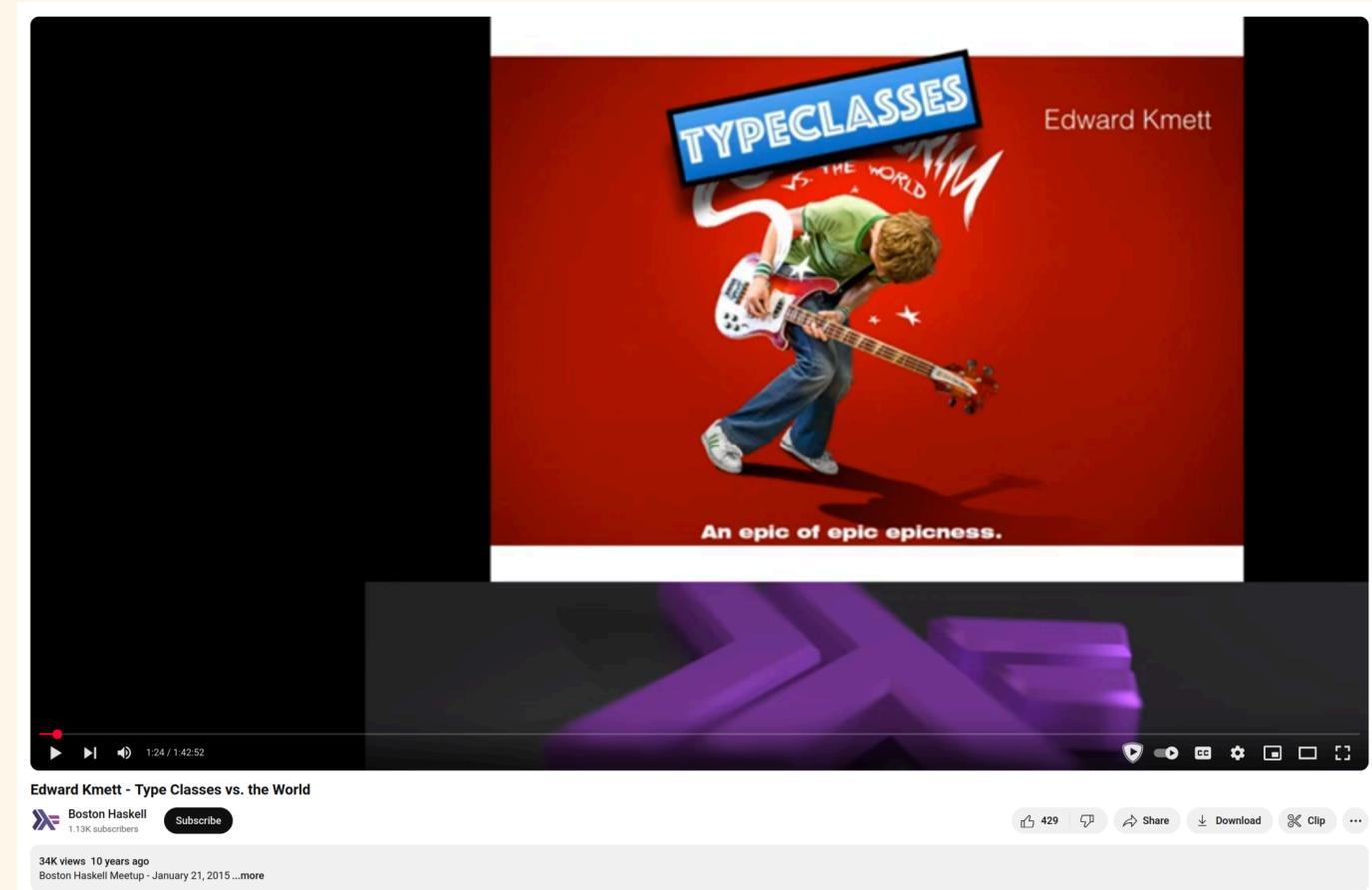
Mixins

Prototypes

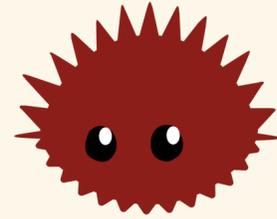
Dependency Injection

Typeclasses vs the World

by Edward Kmett



<https://youtu.be/hIZxTQP1ifo>



Thank You for Listening!



Join us at <https://contextgeneric.dev>