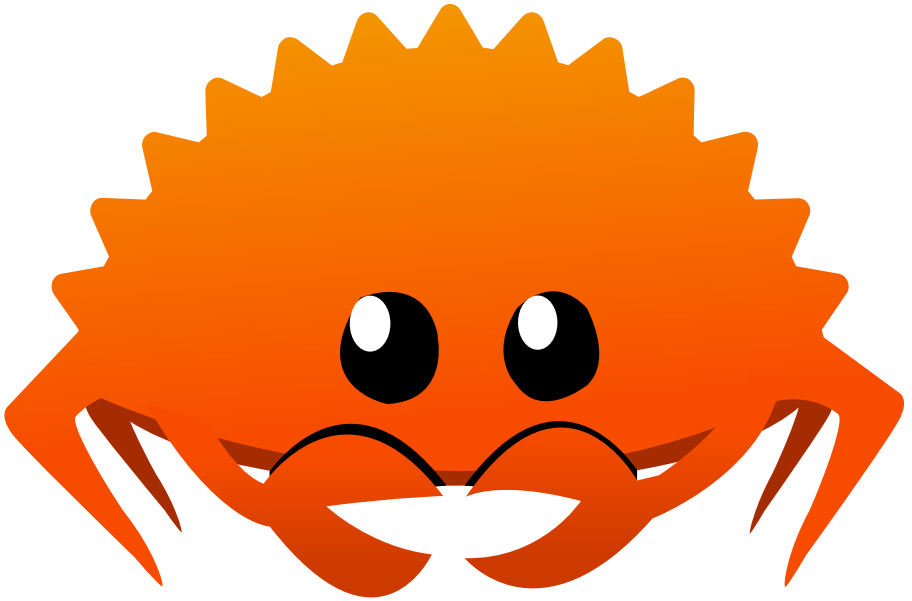
Rust 2024



View slides at https://github.com/nikomatsakis/skillsmatter2022/

by **Nicholas Matsakis**

Who is this guy

 Hi!

Me

- Been working on Rust since 2011
- Co-lead of the Rust language design team

Rust sprouting up all over

… and those are just the foundation platinum sponsors.

What are people doing with Rust?

All kinds of things...

- Networking
- Embedded development
- Kernels, kernel modules
- Blockchain
- CLI apps (ripgrep, just, tokei, ...)
- ...and much more

Why work on Rust?

Why work on Rust?

Why work on Rust?

Why work on Rust?

Rust

A language empowering everyone
to build reliable and efficient software.

GET STARTED

Version 1.61.0

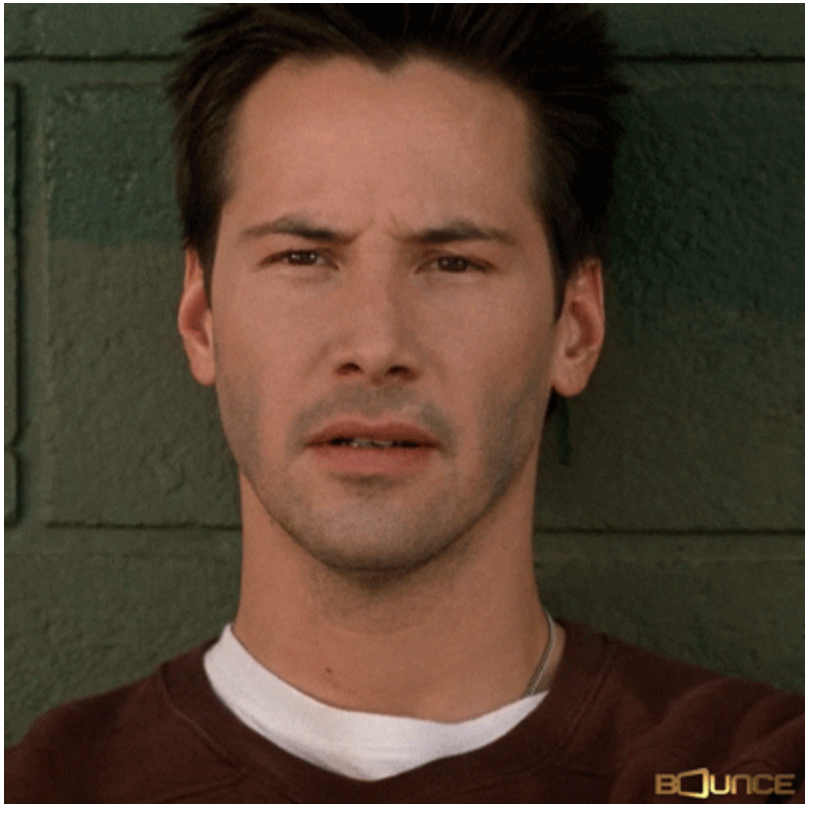
What's Rust's secret sauce?

What's Rust's secret sauce?

A strict and unforgiving type system!

What's Rust's secret sauce?

A strict and unforgiving type system!



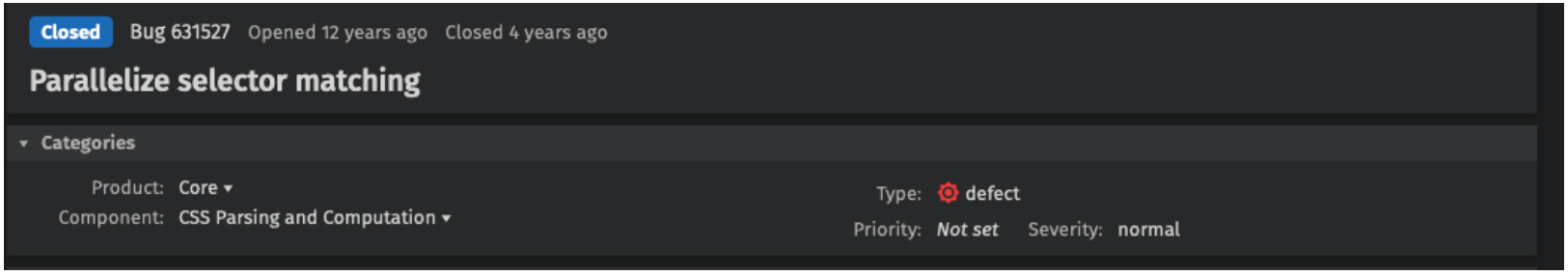
Rust's type system == spinach

Image credit: Clyde Robinson

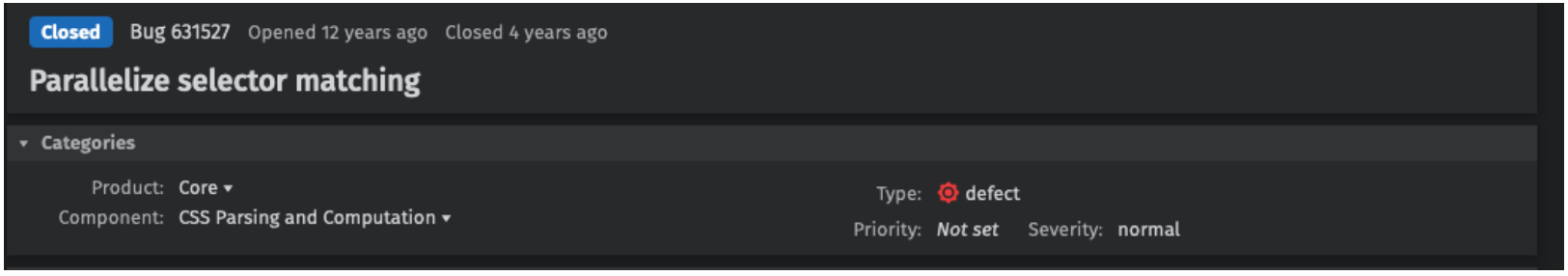
Rust's type system == POPEYE spinach

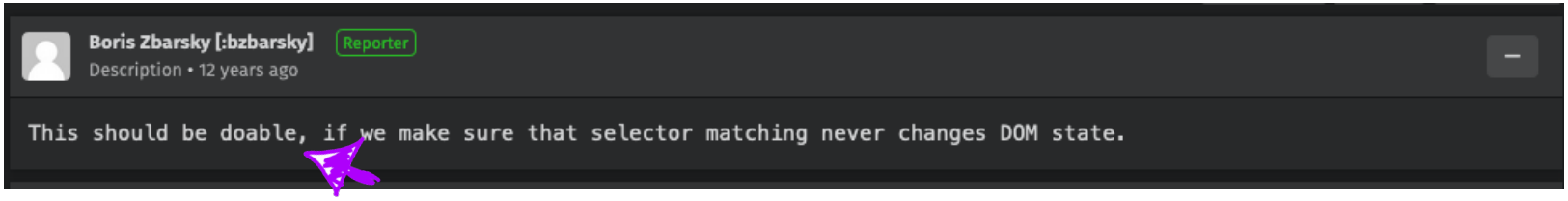
Image credit: Mike Mozart

Example: Mozilla and Stylo

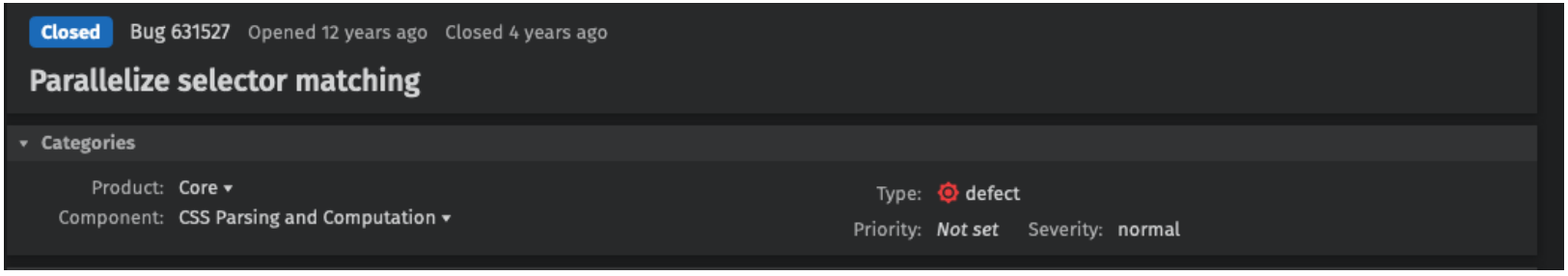


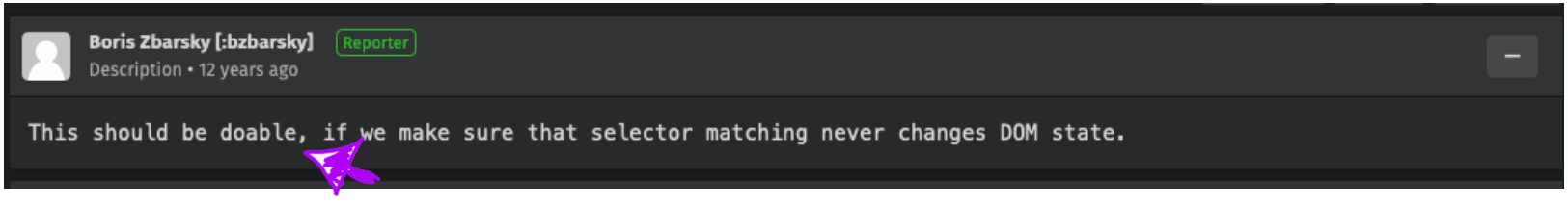
Example: Mozilla and Stylo

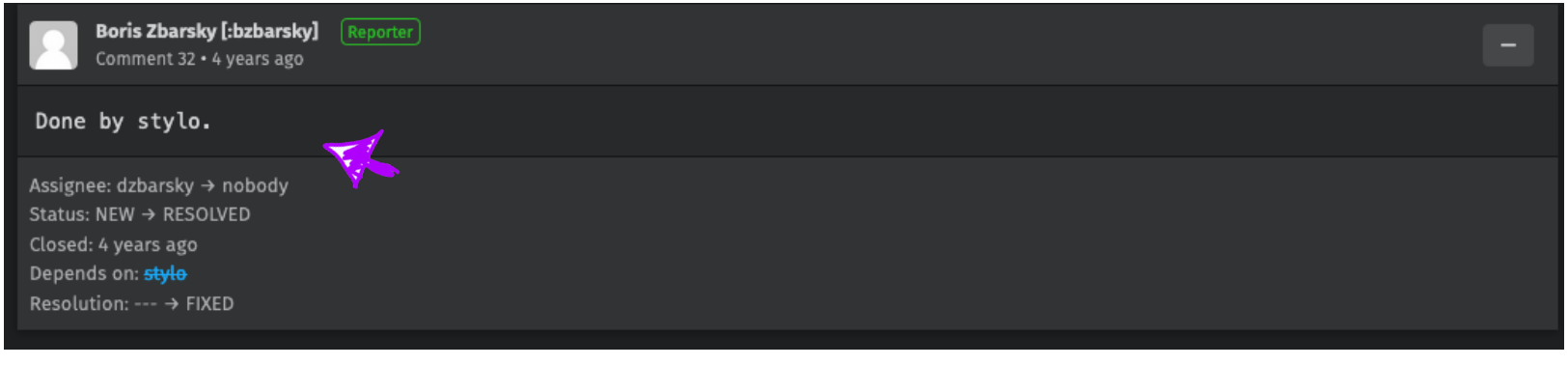




Example: Mozilla and Stylo







Example: Tenable's metrics

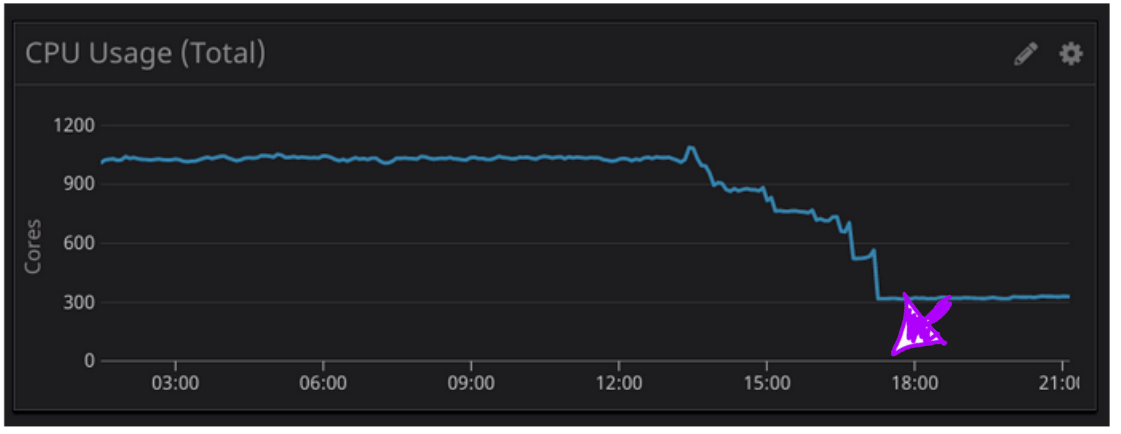
Alan Ning

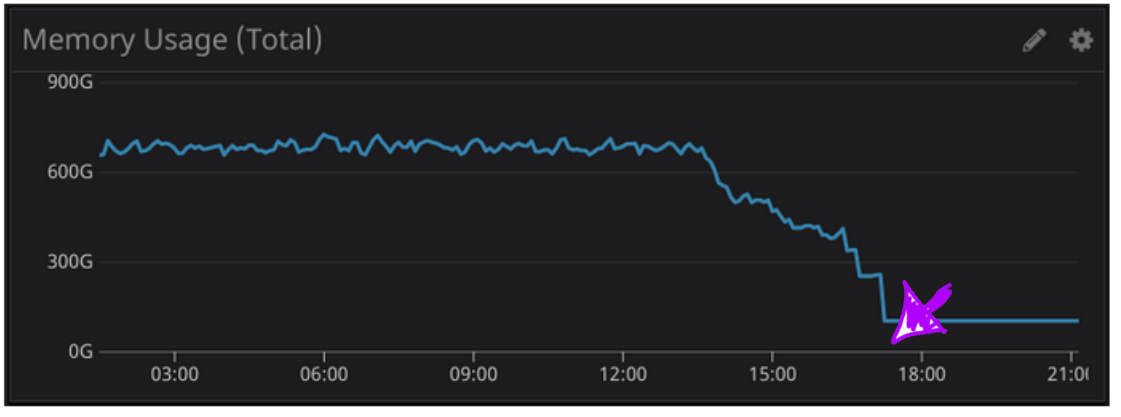
May 6, 2021 · 3 min read · Listen

Optimizing 700 CPUs Away With Rust

[Tenable blog post from 2021-05-06]

Example: Tenable's metrics





CPU / Memory usage dropped drastically following the deployment

Example: Tenable's metrics

> “ With this small change, we were able to optimize away over 700 CPU and 300GB of memory. **This was all implemented, tested and deployed in a single sprint (two weeks).** Once the new filter was deployed, we were able to confirm the resource reduction in Datadog metrics.

Design goals for Rust

⚙️ Reliable	"If it compiles, it works"
🐎 Performant	"idiomatic code runs efficiently"
🥰 Supportive	"the language, tools, and community are here to help"
🧩 Productive	"a little effort does a lot of work"
🔧 Transparent	"you can predict and control low-level details"
🦒 Versatile	"you can do anything with Rust"

Caveat: These are goals that some of us drafted, not official design goals of the Rust project.

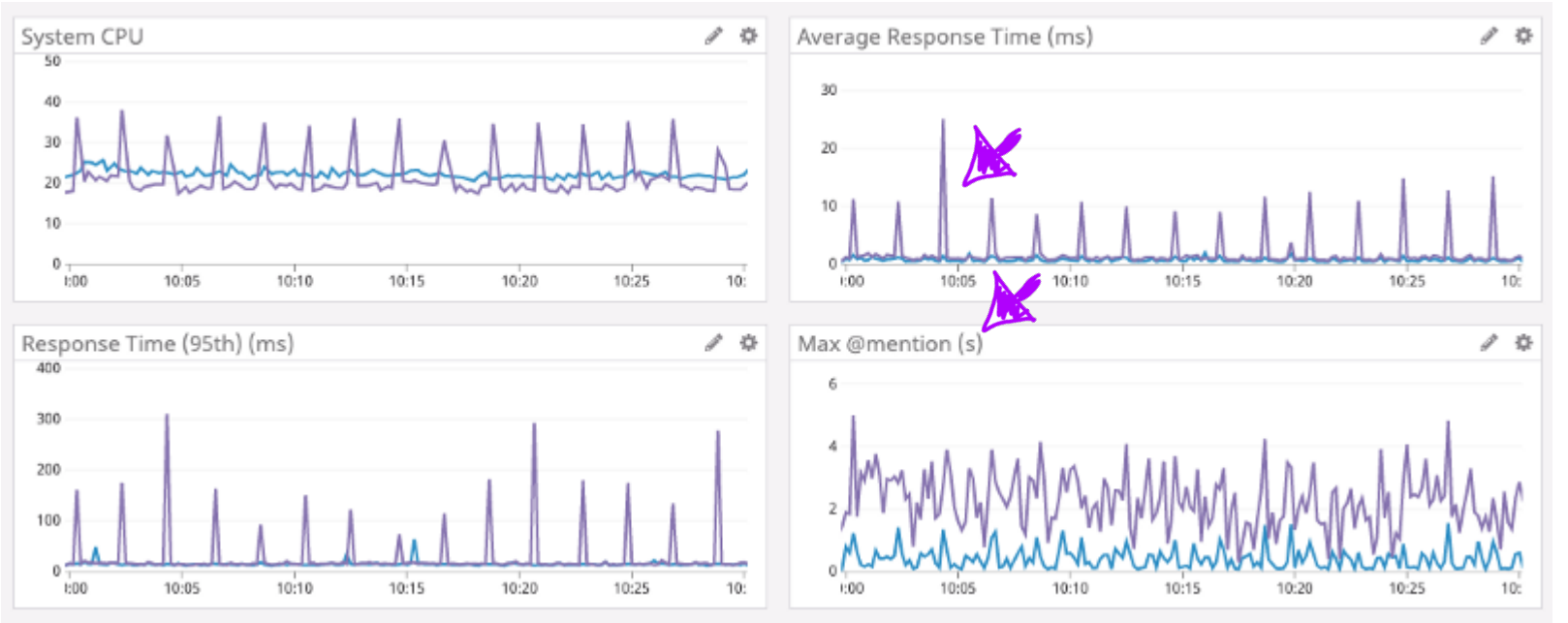
Example: Discord's "read states" service

ENGINEERING & DESIGN

WHY DISCORD IS SWITCHING FROM GO TO RUST

[Discord blogpost from 2022-02-04]

Example: Discord's "read states" service



Example: Discord's "read states" service

> We no longer had to deal with garbage collection, so we figured we could raise the cap of the cache and get even better performance. (...) The results below speak for themselves. **Notice the average time is now measured in microseconds and max @mention is measured in milliseconds**.

Hack without fear

Rust lets you build (and maintain!) the systems you want to build.

Rust 2024

So where do we go from here?

Rust at the start

Artist: Daphne Matsakis

Rust 1.0 released in 2015

Artist: Daphne Matsakis

Rust 2018

Artist: Daphne Matsakis

Rust 2021

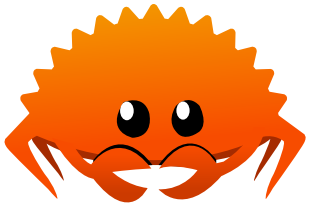
Rust 2024...?

Rust 2024...?

Uh, I don't know. Nodody does, not yet.

Where we are

- If performance and reliability are your top considerations:
  - Rust is your best choice
- If ease of iteration is your top priority:
  - Use a GC'd language like Python, Java, or Go



says:

"A stitch in time saves nine."

Where we are

- If performance and reliability are your top considerations:
  - Rust is your best choice
- If ease of iteration is your top priority:
  - Use a GC'd language like Python, Java, or Go
- **But what about the software in the middle?**

Rust 2024

I think we want a combination of

- Building on our strengths
- Addressing our weaknesses
- Think big opportunities

Building on our strengths

Rust is doing really well in several areas:

- Networking
- Embedded systems, IoT
- Kernels, core architectural layers

Rust in networking, circa 2018

```
async fn process_connection() {
    something().await;
}
```

Rust in networking, circa 2021

- Async fn enables lightweight tasks and a natural coding style...
  - ...but support is missing from many areas of the language, like traits, closures, async-drop.

Rust in networking, circa 2021

Closing the gap requires a number of crates and tools:

- `async_trait` proc macro (shown below)
- `futures` crate combinators
- ...and some things, like async drop, just don't work.

```rust
#[async_trait]
trait AsyncIterator {
    type Item;

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

Rust in networking, circa 2021

- Async fn enables lightweight tasks and a natural coding style...
  - ...but support is missing from many areas of the language, like traits, closures, async-drop.

Rust in networking, circa 2021

- Async fn enables lightweight tasks and a natural coding style...
  - ...but support is missing from many areas of the language, like traits, closures, async-drop.
- Great networking runtimes like tokio, async-std, glommio, embassy, fuschia...
  - ...but no mechanism for interop, leading to a lack of widely used libraries as well as surprising failures.

Rust in networking, circa 2021

- Async fn enables lightweight tasks and a natural coding style...
  - ...but support is missing from many areas of the language, like traits, closures, async-drop.
- Great networking runtimes like tokio, async-std, glommio, embassy, fuschia...
  - ...but no mechanism for interop, leading to a lack of widely used libraries as well as surprising failures.
- Rust developer tooling like cargo, rust-analyzer, rustup is excellent...
  - ...but relatively limited options to debug/profile/test applications, especially async ones.

Rust in networking, circa 2024

- Async fn can be used everywhere: traits, closures, drop
- Rich, interoperable library ecosystem
- Tooling like tokio console to analyze and debug neworked applications
- Works on servers as well as bare-metal environments

How do we get there?

Async vision doc lays out a few key areas:

- Core compiler support for async functions in traits
- Traits for interoperability (read, write, spawn, etc)
- Polish, diagnostics, tooling support

Would you like to help? Join `#wg-async` on rust-lang Zulip.

Building on our strengths

- Networking:
  - Async vision doc
- Embedded, IoT, kernels:
  - Stabilize Rust features that give control over low-level details
  - Take advantage of custom details about a given platform
- General:
  - Rules and tools for unsafe code

Rust 2024

I think we want a combination of

- Building on our strengths ✅
- **Addressing our weaknesses**
- Think big opportunities

Addressing our weaknesses

Rust has some challenges:

- Learning curve
- Cognitive overhead

Journey to loving Rust

Most folks take 3-6 months to learn Rust.

At first, it's ridiculously frustrating.

At some point, you turn the corner, and -- for many of us -- it's hard to imagine using another language.

Key to loving Rust

Learning to *leverage* the Rust type system instead of *fighting* it.

Rust is pushing you towards new patterns. Those patterns are hard to learn, but they are (usually) beneficial.

Detours

But not everybody comes to love Rust.

Some 20% of people on the Rust survey use Rust daily and yet say they "struggle" to be productive.

Why?

Why do people struggle?

Think back to the statue:

Inherent vs **accidental** complexity

Inherent vs accidental complexity

```rust
fn get_lazy(list: &mut Vec<String>) -> &mut String {
    if let Some(s) = list.first_mut() {
        return s;
    }

    list.push(format!("Hello, world!"));
    list.first_mut().unwrap()
}
```

Inherent vs accidental complexity

```rust
fn get_lazy(list: &mut Vec<String>) -> &mut String {
    if let Some(s) = list.first_mut() {
        return s;
    }

    list.push(format!("Hello, world!"));
    list.first_mut().unwrap()
}
```

Inherent complexity: Representing many possibilities

Accidental complexity: `Option` types, `if let` vs `match`

Inherent vs accidental complexity

```rust
fn get_lazy(list: &mut Vec<String>) -> &mut String {
    if let Some(s) = list.first_mut() {
        return s;
    }

    list.push(format!("Hello, world!"));
    list.first_mut().unwrap()
}
```

Inherent complexity: Mutability xor sharing, pointers and references

Accidental complexity: `&mut` syntax

Inherent vs accidental complexity

```rust
fn get_lazy(list: &mut Vec<String>) -> &mut String {
    if let Some(s) = list.first_mut() {
        return s;
    }

    list.push(format!("Hello, world!"));
    list.first_mut().unwrap()
}
```

Inherent complexity: Returning a derived reference

Accidental complexity: Lifetime elision

Inherent vs accidental complexity

```
fn get_lazy(list: &mut Vec<String>) -> &mut String {
    if let Some(s) = list.first_mut() {
        return s;
    }

    list.push(format!("Hello, world!"));
    list.first_mut().unwrap()
}
```

Inherent complexity: Returning a derived reference

Accidental complexity: Lifetime elision

```
fn get_lazy<'a>(list: &'a mut Vec<String>) -> &'a mut String
```

Inherent vs accidental complexity

```
fn get_lazy(list: &mut Vec<String>) -> &mut String {
    if let Some(s) = list.first_mut() {
        return s;
    }

    list.push(format!("Hello, world!"));
    list.first_mut().unwrap()
}
```

Accidental complexity: This code doesn't build!

- `s` was returned from the function, so `s` is borrowed for the rest of the function

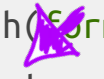
Inherent vs accidental complexity

```
fn get_lazy(list: &mut Vec<String>) -> &mut String {
    if let Some(s) = list.first_mut() {
        return s;
    }

    list.push(format!("Hello, world!"));
    list.first_mut().unwrap()
}
```

- `s` was returned from the function, so `s` is borrowed for the rest of the function
- `s` came from `list`, so `list` is borrowed for the rest of the function too

Inherent vs accidental complexity

```
fn get_lazy(list: &mut Vec<String>) -> &mut String {
    if let Some(s) = list.first_mut() {
        return s;
    }

    list.push(format!("Hello, world!"));
    list.first_mut().unwrap()
}
```



- `s` was returned from the function, so `s` is borrowed for the rest of the function
- `s` came from `list`, so `list` is borrowed for the rest of the function too
- so `push` is illegal

[Try it out]

Workaround

```
fn get_lazy(list: &mut Vec<String>) -> &mut String {
    if !list.is_empty() {
        let s = list.first_mut().unwrap();
        return s;
    }

    list.push(format!("Hello, world!"));
    list.first_mut().unwrap()
}
```

Workaround: move borrow inside the if

Reducing accidental complexity

- Language changes like polonius, implied bounds
- Better environments and materials for learners:
  - Visualize Rust rules
  - Teach borrow checker patterns

Rust 2024

I think we want a combination of

- Building on our strengths ✅
- Addressing our weaknesses ✅
- **Think big opportunities**

Cognitive overhead

Rust makes you care about

- performance
- reliability
- long-term maintenance

...even when you don't want to.

Example: Interfaces

Compare:

```
trait Iterator {
    type Item;

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

with

```
interface Iterator<E> {
    bool hasNext();
    E next();
}
```

Example: Rc vs Arc

Rust has two reference-counted types:

- `Rc<T>`: reference counted -- faster
- `Arc<T>`: *atomic* reference counted -- works across threads

Which should you use?

Example: Async vs not

Earlier we talked about async:

```
trait AsyncIterator {
    type Item;

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

Great to have `AsyncIterator` and `AsyncDrop`, but will we wind up with an `AsyncFoo` for every sync `Foo`?

Think big

Rust is always looking for ways to **eliminate tradeoffs**:

- Can we find a third way that means you don't have to think about it?

Avoiding colors

Maybe instead of defining traits like `AsyncIterator`, we should have `async Iterator`.

Perhaps we can leverage the same mechanism for `const` (compile-time evaluation)?

Can we write "maybe async" code that works in both modes?

Reference.

Rc vs Arc

There are many ways to make reference counting faster:

- Deferred reference counting
- Biased reference counting

Maybe we should try some of them?

Iterative tooling

What if `cargo test` could run tests even when there were compilation errors?

Maybe even skip compiling code that it didn't need?

Unsafe code

Can `cargo test` enforce unsafe code rules by default?

Can we support verifiers and theorem provers, so that people can prove things about unsafe code?

Library with custom errors and lints

diesel#2450

```
let result = diesel::delete(
        scripts
            .filter(id.eq("1"))
    )
    .execute(session.db())
    .map_err(|e| {
        debug!("{:?}", e);
        format!("Could not delete script.")
    });
```

Problem? Using a string, not an integer.

Error?

Library with custom errors and lints

```
the trait bound `diesel::query_builder::SelectStatement<schema::scripts::table,
 diesel::query_builder::select_clause::DefaultSelectClause,
 diesel::query_builder::distinct_clause::NoDistinctClause,
 diesel::query_builder::where_clause::WhereClause<diesel::expression::operators::Eq
 <schema::scripts::columns::id, &str>>>: diesel::query_builder::IntoUpdateTarget` is
not satisfied

the trait `diesel::query_builder::IntoUpdateTarget` is not implemented for
 `diesel::query_builder::SelectStatement<schema::scripts::table,
  diesel::query_builder::select_clause::DefaultSelectClause,
  diesel::query_builder::distinct_clause::NoDistinctClause,
  diesel::query_builder::where_clause::WhereClause<diesel::expression::operators::Eq
  <schema::scripts::columns::id, &str>>>`

help: the following implementations were found:
<diesel::query_builder::SelectStatement<F,
diesel::query_builder::select_clause::DefaultSelectClause,
diesel::query_builder::distinct_clause::NoDistinctClause, W> as
diesel::query_builder::IntoUpdateTarget>rustc(E0277)
```

Platforms

Could

```
#[cfg(unix)]
fn do_something_in_a_unix_way() { }
```

become

```
fn do_something_in_a_unix_way()
where
    std::Platform: Unix,
{
    ...
}
```

Building Rust 2024

I don't exactly know what Rust 2024 will be like.

But I know it's going to be a community effort.

If you're interested in getting involved, take a look at some of the recent blog posts:

- Compiler team ambitions
- Lang team roadmap for 2024
- Library team aspirations

Rust 2024

- Building on our strengths:
  - Async and sync code working at par
  - Stabilize key low-level capabilities
  - Support unsafe code
- Addressing our weaknesses:
  - Smarter analyses, less accidental complexity
  - Developer tooling, documented patterns
- Thinking big:
  - Now's the time!