EDOARDO MORANDI

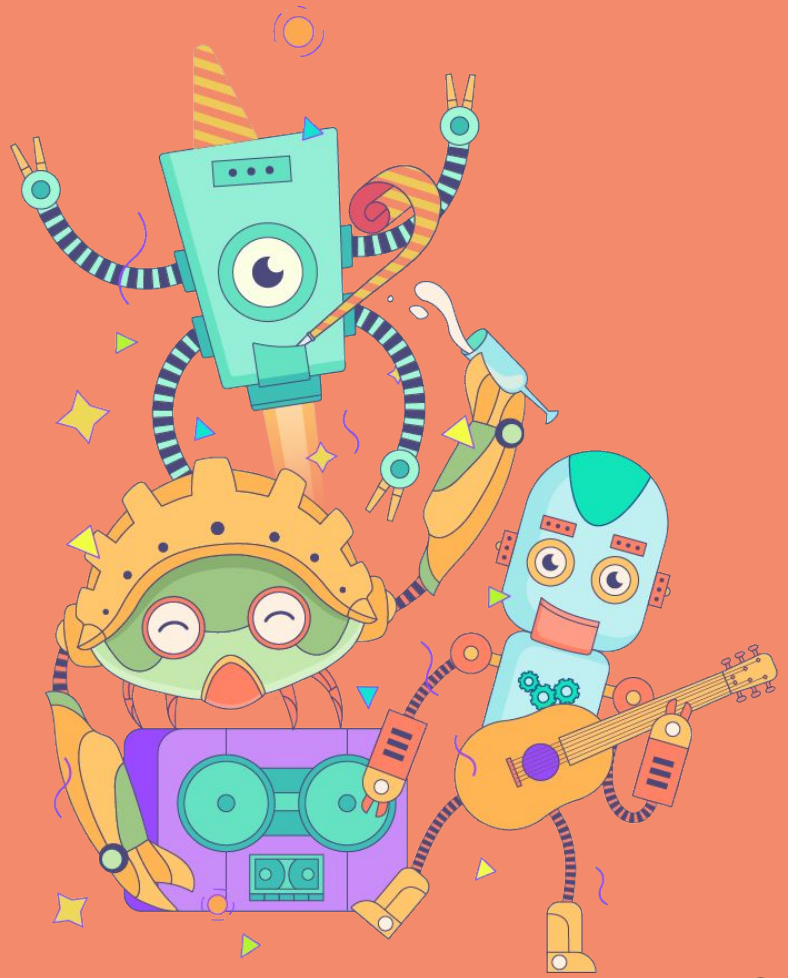
Freelance full-stack developer

Exploring lending async iterators

- The trait

```
trait LendingAsyncIterator {
    type Item<'a>
    where
        Self: 'a;

    fn poll_next<'a>(
        self: Pin<&'a mut Self>,
        cx: &mut task::Context,
    ) -> Poll<Option<Self::Item<'a>>>;
}
```

Questions?



Hello!
I am Edoardo Morandi

- Freelance full-stack developer
- Ex-bioinformatician
- In love with Rust since 2017
- @dodomorandi

What the &#£%§ is a lending async iterator

- Iterator
- Async
- Lending

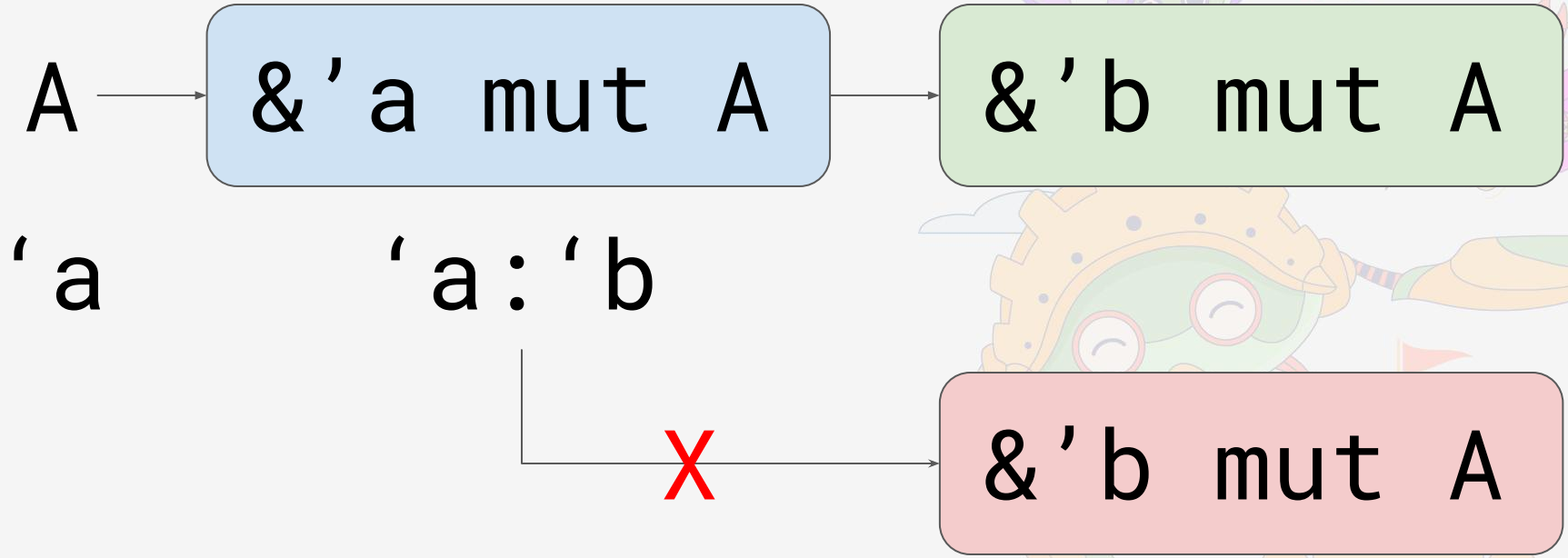
What about lending iterators?

- An interesting topic as well
- Known concept
- Some possible approaches since GAT stabilization (Rust 1.65)

The reasons

- A safe and solid abstraction for resource constrained devices (i.e: bare metal)
- Pushing the limits of the language
- It’s fun 😁

- “Lending” something



- Current ~~AsyncIterator~~ Stream

```rust
trait Stream {
    type Item;

    fn poll_next(
        self: Pin<&mut Self>,
        cx: &mut Context<'_>
    ) -> Poll<Option<Self::Item>>;
}
```

- Current ~~AsyncIterator~~ Stream

```
trait Stream {
    type Item;

    fn poll_next(
        self: Pin<&mut Self>,
        cx: &mut Context<'_>
    ) -> Poll<Option<Self::Item>>;
}
```

- Current ~~AsyncIterator~~ Stream

```
trait Stream {
    type Item;

    fn poll_next(
        self: Pin<&mut Self>,
        cx: &mut Context<'_>
    ) -> Poll<Option<Self::Item>>;
}
```

- Current ~~AsyncIterator~~ Stream

```
trait Stream {
    type Item;

    fn poll_next(
        self: Pin<&mut Self>,
        cx: &mut Context<'_>
    ) -> Poll<Option<Self::Item>>;
}
```

- Current ~~AsyncIterator~~ Stream

```
trait Stream {
    type Item;

    fn poll_next(
        self: Pin<&mut Self>,
        cx: &mut Context<'_>
    ) -> Poll<Option<Self::Item>>;
}
```

- Current ~~AsyncIterator~~ Stream

```rust
trait Stream {
    type Item;

    fn poll_next(
        self: Pin<&mut Self>,
        cx: &mut Context<'_>
    ) -> Poll<Option<Self::Item>>;
}
```

- Current ~~AsyncIterator~~ Stream

```rust
trait Stream {
    type Item;

    fn poll_next(
        self: Pin<&mut Self>,
        cx: &mut Context<'_>
    ) -> Poll<Option<Self::Item>>;
}
```

- Back to the trait

```
trait LendingAsyncIterator {
    type Item<'a>
    where
        Self: 'a;

    fn poll_next<'a>(
        self: Pin<&'a mut Self>,
        cx: &mut task::Context<'_>,
    ) -> Poll<Option<Self::Item<'a>>>;
}
```

- Back to the trait

```
trait LendingAsyncIterator {
    type Item<'a>
    where
        Self: 'a;

    fn poll_next<'a>(
        self: Pin<&'a mut Self>,
        cx: &mut task::Context<'_>,
    ) -> Poll<Option<Self::Item<'a>>>;
}
```

- Back to the trait

```
trait LendingAsyncIterator {
    type Item<'a>
    where
        Self: 'a;

    fn poll_next<'a>(
        self: Pin<&'a mut Self>,
        cx: &mut task::Context<'_>,
    ) -> Poll<Option<Self::Item<'a>>>;
}
```

- Back to the trait

```
trait LendingAsyncIterator {
    type Item<'a>
    where
        Self: 'a;

    fn poll_next<'a>(
        self: Pin<&'a mut Self>,
        cx: &mut task::Context<'_>,
    ) -> Poll<Option<Self::Item<'a>>>;
}
```

- Back to the trait

```
trait LendingAsyncIterator {
    type Item<'a>
    where
        Self: 'a;

    fn poll_next<'a>(
        self: Pin<&'a mut Self>,
        cx: &mut task::Context<'_>,
    ) -> Poll<Option<Self::Item<'a>>>;
}
```

- Back to the trait

```
trait LendingAsyncIterator {
    type Item<'a>
    where
        Self: 'a;

    fn poll_next<'a>(
        self: Pin<&'a mut Self>,
        cx: &mut task::Context<'_>,
    ) -> Poll<Option<Self::Item<'a>>>;
}
```

- Back to the trait

```
trait LendingAsyncIterator {
    type Item<'a>
    where
        Self: 'a;

    fn poll_next<'a>(
        self: Pin<&'a mut Self>,
        cx: &mut task::Context<'_>,
    ) -> Poll<Option<Self::Item<'a>>>;
}
```

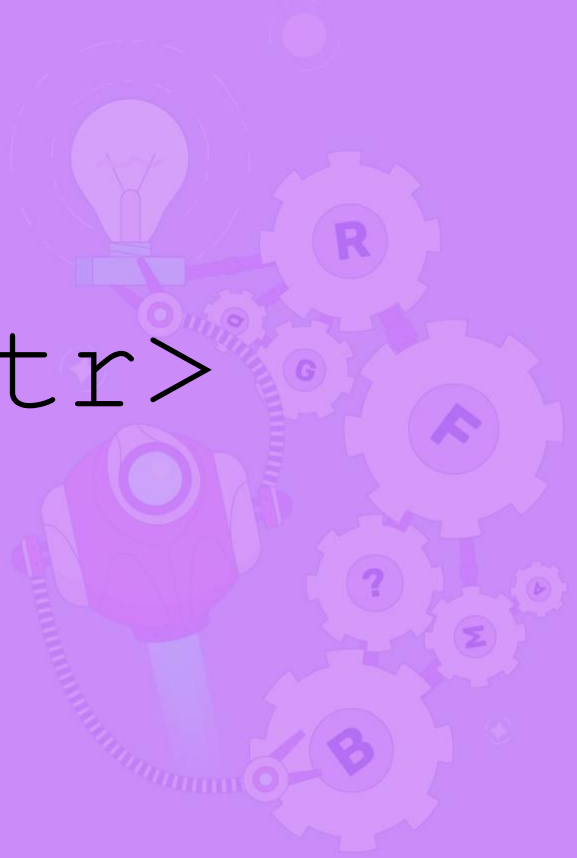
- Back to the trait

```
trait LendingAsyncIterator {
    type Item<'a>
    where
        Self: 'a;

    fn poll_next<'a>(
        self: Pin<&'a mut Self>,
        cx: &mut task::Context<'_>,
    ) -> Poll<Option<Self::Item<'a>>>;
}
```

- Back to the trait

```
trait LendingAsyncIterator {
    type Item<'a>
    where
        Self: 'a;

    fn poll_next<'a>(
        self: Pin<&'a mut Self>,
        cx: &mut task::Context<'_>,
    ) -> Poll<Option<Self::Item<'a>>>;
}
```

`std::pin::Pin<Ptr>`



```
::std
::pin
::Pin<Ptr>
```

- Wrapper around pointer-like types (i.e.: &mut T, Box<T>, Arc<T>)
- Ptr: Deref/DerefMut
- Unwrap if Ptr::Target: Unpin
- Unpin = auto-trait
- !Unpin

```
::std
::pin
::Pin<Ptr>
```

Ptr::Target: !Unpin -> No Unwrap

Pin and structs

```
struct A {
    value: i32,
    b: B,
}

struct B {
    value: i16,
    _marker: std::marker::PhantomPinned,
}
```

Pin and structs

```
impl Future for A {
    type Output = (i32, i16);

    fn poll(
        self: Pin<&mut Self>,
        cx: &mut task::Context<'_>,
    ) -> Poll<Self::Output> {
        let a = self.get_mut();


        todo!()
    }
}
```

Pin and structs

```
impl Future for A {
    type Output = (i32, i16);

    fn poll(
        self: Pin<&mut Self>,
        cx: &mut task::Context<'_>,
    ) -> Poll<Self::Output> {
        let a = self.get_mut();


        todo!()
    }
}
```

Pin and structs

```
impl Future for A {
    type Output = (i32, i16);

    fn poll(
        self: Pin<&mut Self>,
        cx: &mut task::Context<'_>,
    ) -> Poll<Self::Output> {
        let a = self.get_mut();
                    ^^^^^^^^ within `A`, the trait `Unpin`
is not implemented for `PhantomPinned`, which is required
by `A: Unpin`
        todo!()
    }
}
```

Pin and structs

```rust
fn poll(...) -> Poll<Self::Output> {
    // SAFETY: we are not going to move anything
    let a = unsafe { self.get_unchecked_mut() };
    a.value += 1
    a.b.value *= 2;

    Poll::Ready((a.value, a.b.value))
}
```

Pin and structs – safely

```rust
#[pin_project]
struct A {
    value: i32,
    #[pin]
    b: B,
}

#[pin_project]
struct B {
    value: i16,
    _marker: std::marker::PhantomPinned,
}
```

- Pin and structs – safely

```
fn poll(...) -> Poll<Self::Output> {
    let a = self.project();
    *a.value += 1;

    let b = a.b.project();
    *b.value *= 2;

    Poll::Ready((*a.value, *b.value))
}
```

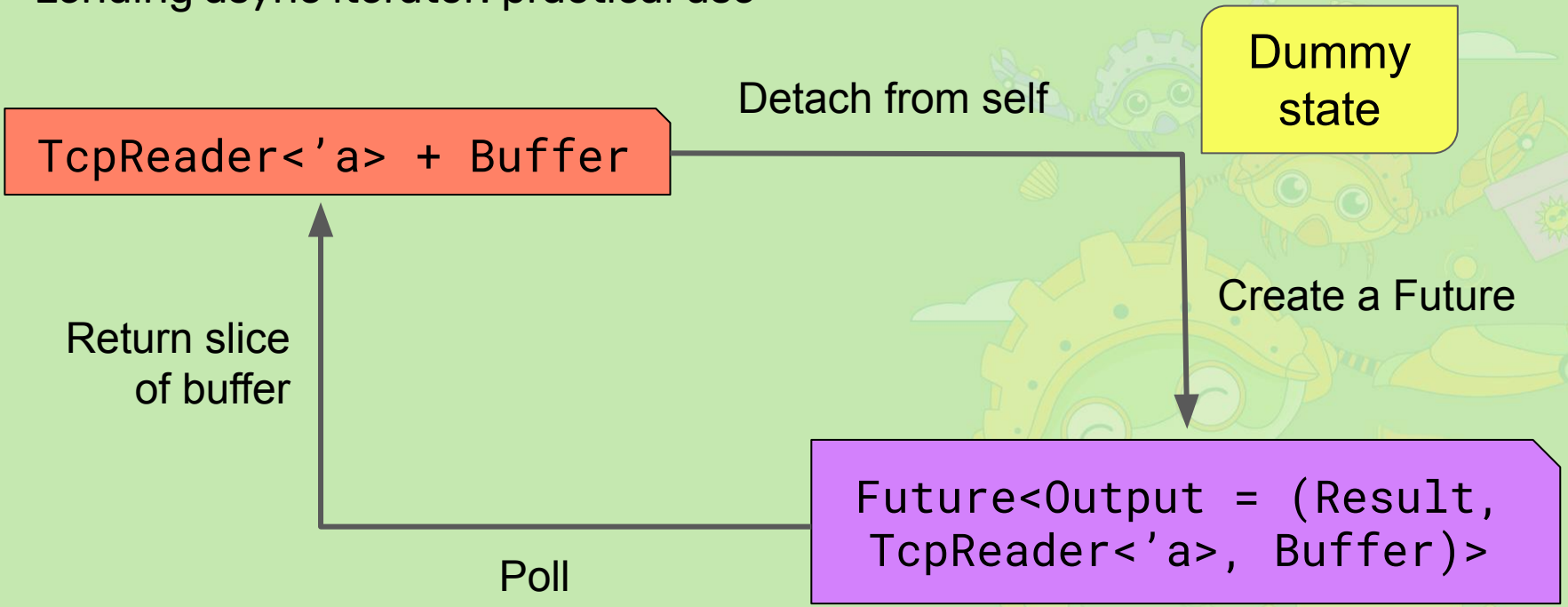
Reading a TcpReader struct in Embassy, without allocating

- Lending async iterator: practical use

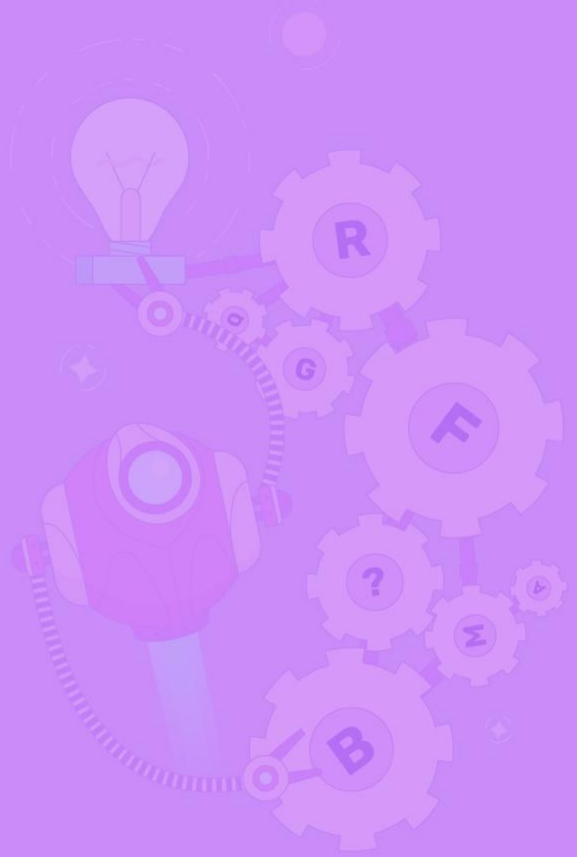
1. Allocate a fixed array buffer on the stack
2. TcpReader::read into the buffer
3. Check if returned bytes is zero to stop
4. Use the returned bytes to take a slice from the buffer
5. Use the slice
6. Go back to step 2

There is no stream/async iterator abstraction here, just a raw loop!

- Lending async iterator: practical use



Using the lending async iterator



Same issue of `Iterator`, `Future` and `Stream`

- Only the polling/next function is bad DX
- Additional “helper” methods
- Directly or using an extension trait

Emerging properties

- Cannot “store” elements
- Some methods cannot exist (i.e.: `collect`, `last`, `max`, `min`...)

A simple helper method

```rust
fn for_each<F>(self, f: F) -> ForEach<Self, F>
where
    Self: Sized,
    for<'a> F: FnMut(Self::Item<'a>),
{
    ForEach {
        async_iter: self,
        f,
    }
}
```

A simple helper method

```rust
#[pin_project]
pub struct ForEach<I, F> {
    #[pin]
    async_iter: I,
    f: F,
}
```

A simple helper method

```
impl<I, F> Future for ForEach<I, F>
where
    I: LendingAsyncIterator,
    for<'a> F: FnMut(I::Item<'a>),
{
    type Output = ();

    fn poll /* ... */ {

    }
}
```

A simple helper method

```rust
fn poll(/* ... */) -> Poll<Self::Output> {
    let mut this = self.project();
    while let Some(element) = task::ready!(
        this.async_iter.as_mut().poll_next(cx)
    ) {
        (this.f)(element);
    }
    Poll::Ready(())
}
```

A simple helper method

```
fn poll(/* ... */) -> Poll<Self::Output> {
    let mut this = self.project();
    while let Some(element) = task::ready!(
        this.async_iter.as_mut().poll_next(cx)
    ) {
        (this.f)(element);
    }
    Poll::Ready(())
}
```

A simple helper method

```
fn poll(/* ... */) -> Poll<Self::Output> {
    let mut this = self.project();
    while let Some(element) = task::ready!(
        this.async_iter.as_mut().poll_next(cx)
    ) {
        (this.f)(element);
    }
    Poll::Ready(())
}
```

A simple helper method

```
fn poll(/* ... */) -> Poll<Self::Output> {
    let mut this = self.project();
    while let Some(element) = task::ready!(
        this.async_iter.as_mut().poll_next(cx)
    ) {
        (this.f)(element);
    }
    Poll::Ready(())
}
```

A simple helper method

```
fn poll(/* ... */) -> Poll<Self::Output> {
    let mut this = self.project();
    while let Some(element) = task::ready!(
        this.async_iter.as_mut().poll_next(cx)
    ) {
        (this.f)(element);
    }
    Poll::Ready(())
}
```

A simple helper method

```
fn poll(/* ... */) -> Poll<Self::Output> {
    let mut this = self.project();
    while let Some(element) = task::ready!(
        this.async_iter.as_mut().poll_next(cx)
    ) {
        (this.f)(element);
    }
    Poll::Ready(())
}
```

A simple helper method

```
fn poll(/* ... */) -> Poll<Self::Output> {
    let mut this = self.project();
    while let Some(element) = task::ready!(
        this.async_iter.as_mut().poll_next(cx)
    ) {
        (this.f)(element);
    }
    Poll::Ready(())
}
```

A simple helper method

```
fn poll(/* ... */) -> Poll<Self::Output> {
    let mut this = self.project();
    while let Some(element) = task::ready!(
        this.async_iter.as_mut().poll_next(cx)
    ) {
        (this.f)(element);
    }
    Poll::Ready(())
}
```

A simple helper method

```rust
fn poll(/* ... */) -> Poll<Self::Output> {
    let mut this = self.project();
    while let Some(element) = task::ready!(
        this.async_iter.as_mut().poll_next(cx)
    ) {
        (this.f)(element);
    }
    Poll::Ready(())
}
```

Let's try `Filter`...

```rust
fn filter<P>(self, predicate: P) -> Filter<Self, P>
where
    Self: Sized,
    for<'a> P: FnMut(&Self::Item<'a>) -> bool,
{
    Filter {
        async_iter: self,
        predicate,
    }
}
```

The `Filter` struct

```rust
#[pin_project]
pub struct Filter<I, P> {
    #[pin]
    async_iter: I,
    predicate: P,
}
```

Using the lending async iterator

impl `LendingAsyncIterator` for `Filter`

```rust
impl<I, P> LendingAsyncIterator for Filter<I, P>
where
    I: LendingAsyncIterator,
    for<'a> P: FnMut(&I::Item<'a>) -> bool,
{
    type Item<'a> = I::Item<'a>
    where
        Self: 'a;
    /* ... */
}
```

Using the lending async iterator

impl `LendingAsyncIterator` for `Filter`

```
fn poll_next<'a>(/* ... */) -> /* ... */ {
    let mut this = self.project();
    while let Some(element) = task::ready!(
        this.async_iter.as_mut().poll_next(cx)
    ) {
        if (this.predicate)(&element) {
            return Poll::Ready(Some(element));
        }
    }
    Poll::Ready(None)
}
```

Using the lending async iterator

impl `LendingAsyncIterator` for `Filter`

```
fn poll_next<'a>(/* ... */) -> /* ... */ {
    let mut this = self.project();
    while let Some(element) = task::ready!(
        this.async_iter.as_mut().poll_next(cx)
    ) {
        if (this.predicate)(&element) {
            return Poll::Ready(Some(element));
        }
    }
    Poll::Ready(None)
}
```

Using the lending async iterator

impl `LendingAsyncIterator` for `Filter`

```
fn poll_next<'a>(/* ... */) -> /* ... */ {
    let mut this = self.project();
    while let Some(element) = task::ready!(
        this.async_iter.as_mut().poll_next(cx)
    ) {
        if (this.predicate)(&element) {
            return Poll::Ready(Some(element));
        }
    }
    Poll::Ready(None)
}
```

Using the lending async iterator

impl `LendingAsyncIterator` for `Filter`

```
fn poll_next<'a>(/* ... */) -> /* ... */ {
    let mut this = self.project();
    while let Some(element) = task::ready!(
        this.async_iter.as_mut().poll_next(cx)
    ) {
        if (this.predicate)(&element) {
            return Poll::Ready(Some(element));
        }
    }
    Poll::Ready(None)
}
```

Using the lending async iterator

impl `LendingAsyncIterator` for `Filter`

```
fn poll_next<'a>(/* ... */) -> /* ... */ {
    let mut this = self.project();
    while let Some(element) = task::ready!(
        this.async_iter.as_mut().poll_next(cx)
    ) {
        if (this.predicate)(&element) {
            return Poll::Ready(Some(element));
        }
    }
    Poll::Ready(None)
}
```

Using the lending async iterator

impl `LendingAsyncIterator` for `Filter`

```
fn poll_next<'a>(/* ... */) -> /* ... */ {
    let mut this = self.project();
    while let Some(element) = task::ready!(
        this.async_iter.as_mut().poll_next(cx)
    ) {
        if (this.predicate)(&element) {
            return Poll::Ready(Some(element));
        }
    }
    Poll::Ready(None)
}
```

Using the lending async iterator

impl `LendingAsyncIterator` for `Filter`

```
fn poll_next<'a>(/* ... */) -> /* ... */ {
    let mut this = self.project();
    while let Some(element) = task::ready!(
        this.async_iter.as_mut().poll_next(cx)
    ) {
        if (this.predicate)(&element) {
            return Poll::Ready(Some(element));
        }
    }
    Poll::Ready(None)
}
```

Using the lending async iterator

impl `LendingAsyncIterator` for `Filter`

```
fn poll_next<'a>(/* ... */) -> /* ... */ {
    let mut this = self.project();
    while let Some(element) = task::ready!(
        this.async_iter.as_mut().poll_next(cx)
    ) {
        if (this.predicate)(&element) {
            return Poll::Ready(Some(element));
        }
    }
    Poll::Ready(None)
}
```

Excuse me?

```
error[E0499]: cannot borrow
`this.async_iter` as mutable
more than once at a time
```

Excuse me?

```
fn poll_next<'a>(
                -- lifetime `'a` defined here
this.async_iter.as_mut().poll_next(cx)
^^^^^^^^^^^^^^^^ `this.async_iter` was mutably borrowed here in
the previous iteration of the loop
return Poll::Ready(Some(element));
       -------------------------- returning this value requires
that `this.async_iter` is borrowed for `'a`
```

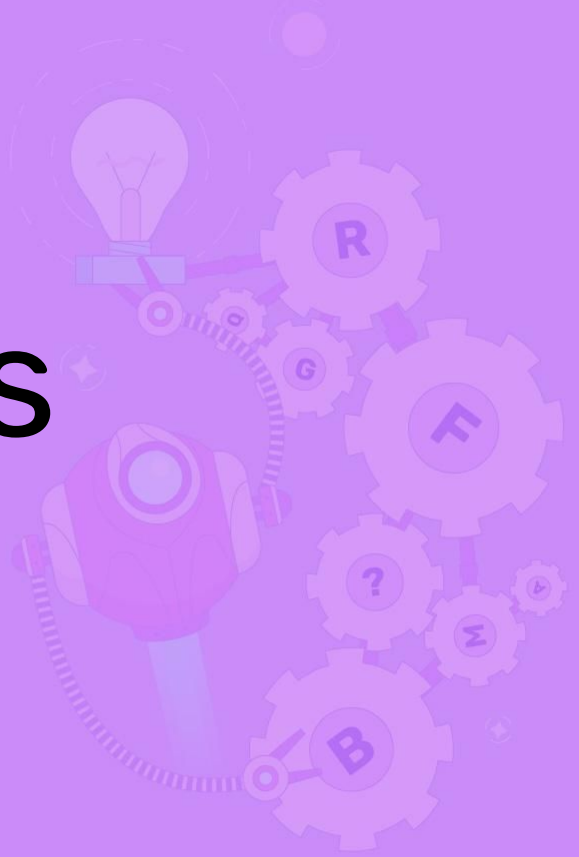
Excuse me?????

```rust
fn poll_next<'a>(/* ... */) -> /* ... */ {
    let mut this = self.project();
    while let Some(element) = task::ready!(
        this.async_iter.as_mut().poll_next(cx)
    ) {
        if (this.predicate)(&element) {
            return Poll::Ready(Some(element));
        }
    }
    Poll::Ready(None)
}
```

Excuse me?????

```
fn poll_next<'a>(/* ... */) -> /* ... */ {
    let mut this = self.project();
    while let Some(element) = task::ready!(
        this.async_iter.as_mut().poll_next(cx)
    ) {
        if (this.predicate)(&element) {
            return Poll::Ready(Some(element));
        }
    }
    Poll::Ready(None)
}
```

-Z polonius



- Mmmmmhhhhh…

```
error[E0515]: cannot return
value referencing local data
`this.async_iter`
```

Mmmmmhhhhh…

```
this.async_iter.as_mut().poll_next(cx)
---------------- `this.async_iter` is borrowed here
return Poll::Ready(Some(element));
       ^^^^^^^^^^^^^^^^^^^^^^^^^^ returns a value
referencing data owned by the current function
```

Using the lending async iterator

Pin::as_mut

```rust
impl<Ptr: DerefMut> Pin<Ptr> {
    fn as_mut(&mut self)
        -> Pin<&mut Ptr::Target>;
}
```

Using the lending async iterator

- `Pin::as_mut`

```rust
fn test<T>(mut a: Pin<&'t mut T>) {
    let b = a.as_mut();
}
```

Using the lending async iterator

- `Pin::as_mut`

```
fn test<T>(mut a: Pin<&'t mut T>) {
    let b = a.as_mut();
}
```

Lifetime `'a -'t: 'a`

Using the lending async iterator

- `Pin::as_mut`

```
fn test<T>(mut a: Pin<&'t mut T>) {
    let b = a.as_mut();
}
```

```
Pin::as_mut(&'a mut a)
```

Using the lending async iterator

Pin::as_mut

```rust
impl<Ptr: DerefMut> Pin<Ptr> {
    fn as_mut(&mut self)
        -> Pin<&mut Ptr::Target>;
}
```

Using the lending async iterator

- `Pin::as_mut`

```
fn test<T>(mut a: Pin<&'t mut T>) {
    let b = a.as_mut();
}
```

`Pin<&'a mut T>` (not ‘t)

Pin::as_mut

```rust
fn test<T>(mut a: Pin<&mut T>) -> Pin<&mut T> {
    let b = a.as_mut();
    b
}
```

```
error[E0515]: cannot return value referencing
function parameter `a`
```

The magic of reborrows

```
fn test<T>(a: &mut T) -> &mut T {
    use_mut_ref(a);
    use_mut_ref(a);
    let b = &mut *a;
    b
}
```

This just works™

This will (hopefully) improve

```rust
#![feature(pin_ergonomics)]
#![allow(incomplete_features)]

while let Some(element) = task::ready!(
    /* look ma', no as_mut()! */
    this.async_iter.poll_next(cx)
) {
    if (this.predicate)(&element) {
        return Poll::Ready(Some(element));
    }
}
```

Back to the impl

```rust
fn poll_next<'a>(/* ... */) -> /* ... */ {
    let mut this = self.project();
    while let Some(element) = task::ready!(
        this.async_iter.as_mut().poll_next(cx)
    ) {
        if (this.predicate)(&element) {
            return Poll::Ready(Some(element));
        }
    }
    Poll::Ready(None)
}
```

Workaround using unsafe

```
let this = self.project();
let predicate = this.predicate;

// SAFETY: we only use the reference once
re-pinned
let async_iter = unsafe {
    this.async_iter.get_unchecked_mut()
};
```

Workaround using unsafe

```rust
// SAFETY: it was pinned before, we pin it again
while let Some(element) =
    task::ready!(unsafe {
        Pin::new_unchecked(&mut *async_iter)
    }.poll_next(cx))
{
    if (predicate)(&element) {
        return Poll::Ready(Some(element));
    }
}
Poll::Ready(None)
```

Workaround using unsafe

```
// SAFETY: it was pinned before, we pin it again
while let Some(element) =
    task::ready!(unsafe {
        Pin::new_unchecked(&mut *async_iter)
    }.poll_next(cx))
{
    if (predicate)(&element) {
        return Poll::Ready(Some(element));
    }
}
Poll::Ready(None)
```

Workaround using unsafe

```
// SAFETY: it was pinned before, we pin it again
while let Some(element) =
    task::ready!(unsafe {
        Pin::new_unchecked(&mut *async_iter)
    }.poll_next(cx))
{
    if (predicate)(&element) {
        return Poll::Ready(Some(element));
    }
}
Poll::Ready(None)
```

Workaround using unsafe

```rust
// SAFETY: it was pinned before, we pin it again
while let Some(element) =
    task::ready!(unsafe {
        Pin::new_unchecked(&mut *async_iter)
    }.poll_next(cx))
{
    if (predicate)(&element) {
        return Poll::Ready(Some(element));
    }
}
Poll::Ready(None)
```

It works!

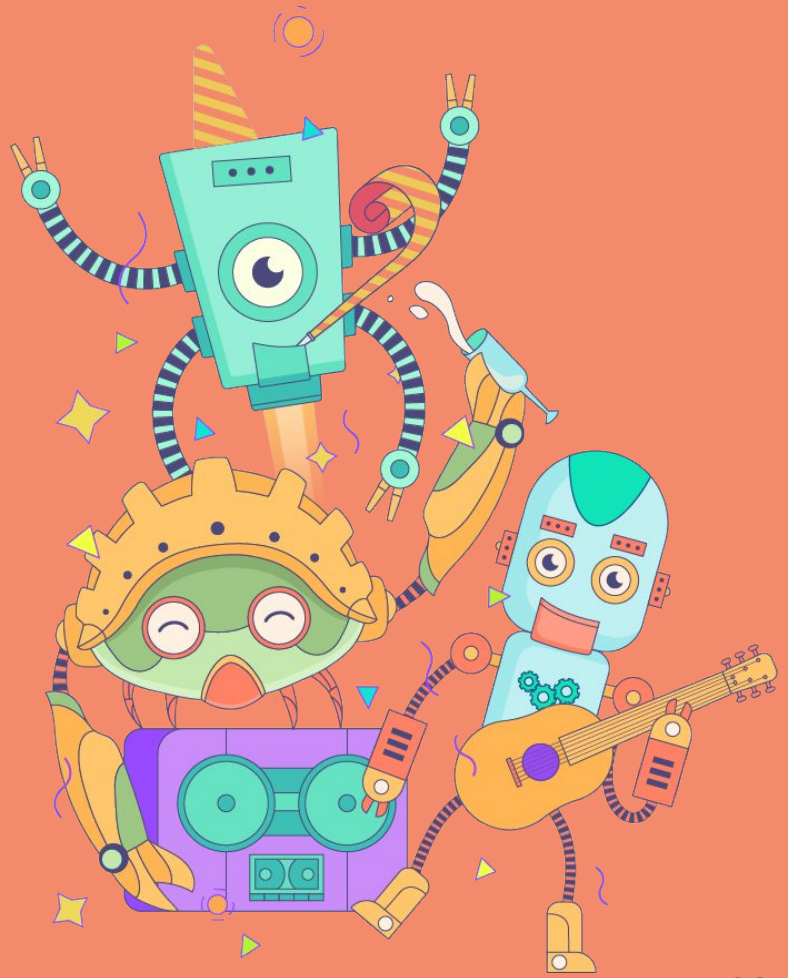
- *Tested with Miri, everything is fine*
- *Full knowledge of what you are doing, hard to understand the real issues*
- *Still requires Polonius*

Time’s out!

Conclusions

- Basic idea of *async lending iterators* as bleeding-edge powerful abstraction
- Experiment with unstable features and unsafe (but stay away from production!)
- Try to reason with the compiler (not against it), and remember to have fun
- Looooooots of dark corners we did not have to look at (i.e.: async closures)

Thank you!



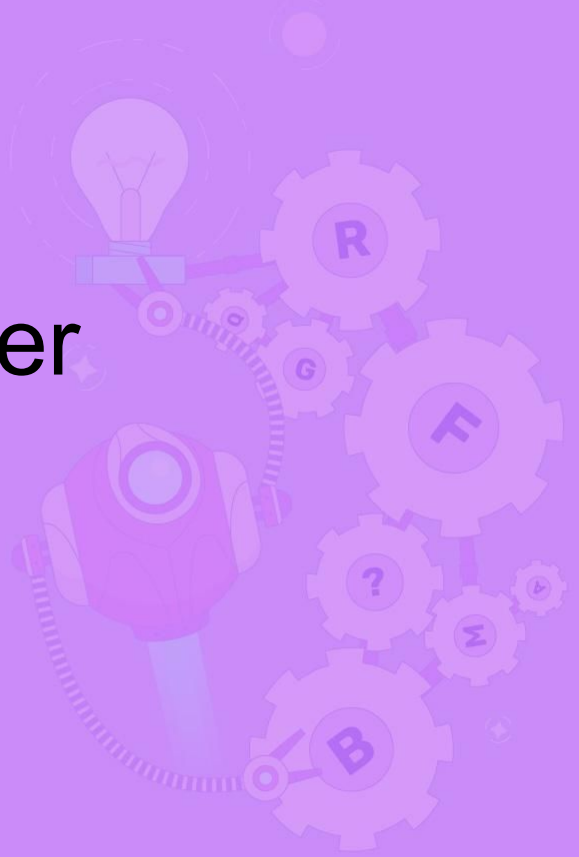
Edoardo Morandi

morandidodo@gmail.com

gh: dodomorandi/lending-async-iterators-rustlab2024

Bonus slides

Embassy TcpReader



Impl highlights (1)

```rust
#[pin_project(
    project = TcpReaderLendingAsyncIteratorProj,
)]
pub struct TcpReaderLendingAsyncIterator<
    'd,
    const BUF_SIZE: usize,
> {
    #[pin]
    status: Status<'d, BUF_SIZE>,
}
```

Impl highlights (1)

```rust
#[pin_project(
    project = TcpReaderOrReadFutureProj,
    project_replace = TcpReaderOrReadFutureProjOwn,
)]
enum Status<'d, const BUF_SIZE: usize> {
    Data {
        tcp_reader: TcpReader<'d>,
        buffer: [u8; BUF_SIZE],
    },
    Future(#[pin] TcpReaderReadFuture<'d, BUF_SIZE>),
    Invalid,
}
```

Impl highlights (2)

```
match this.status.as_mut().project() {
    TcpReaderOrReadFutureProj::Data { .. } => {
        let TcpReaderOrReadFutureProjOwn::Data {
            tcp_reader,
            buffer,
        } =
            this.status.as_mut().project_replace(Status::Invalid)
        else {
            unreachable!();
        };
        /* ... */
    }
}
```

Impl highlights (3)

```
let future = tcp_reader_read_future::create(
    tcp_reader,
    buffer,
);
this.status.set(Status::Future(future));
```

Impl highlights (3) (hello TAIT!)

```rust
pub type TcpReaderReadFuture<
    'd,
    const BUF_SIZE: usize,
> = impl Future<
    Output = (
        Result<usize, embassy_net::tcp::Error>,
        TcpReader<'d>,
        [u8; BUF_SIZE],
    ),
>;
```

Impl highlights (3)

```rust
pub(super) fn create<const BUF_SIZE: usize>(
    mut tcp_reader: TcpReader<'_>,
    mut buffer: [u8; BUF_SIZE],
) -> TcpReaderReadFuture<'_, BUF_SIZE> {
    async move {
        let result = tcp_reader.read(
            &mut buffer,
        ).await;
        (result, tcp_reader, buffer)
    }
}
```

Impl highlights (4)

```
TcpReaderOrReadFutureProj::Data { .. } => {
    /*..*/
    let TcpReaderOrReadFutureProj::Future(future) =
        this.status.project()
    else {
        unreachable!();
    };
    let (result, tcp_reader, buffer) = ready!(future.poll(cx));
    self.handle_future_result(result, tcp_reader, buffer)
}
```

Impl highlights (4)

```
TcpReaderOrReadFutureProj::Future(future) => {
    let (result, tcp_reader, buffer) =
        ready!(future.poll(cx));
    self.handle_future_result(
        result,
        tcp_reader,
        buffer,
    )
}
```

Impl highlights (4)

```
TcpReaderOrReadFutureProj::Invalid => unreachable!(),
```

Impl highlights (4+5)

```
fn handle_future_result(/* ... */) -> /* ... */ {
    let mut this = self.project();
    this.status.set(Status::Data { tcp_reader, buffer });
    let TcpReaderOrReadFutureProj::Data { buffer, .. } =
        this.status.project()
    else {
        unreachable!()
    };
    /* ... */
}
```

Impl highlights (4+5)

```
Poll::Ready(
    result
        .map(|bytes_read|
            (bytes_read != 0)
                .then(|| &buffer[..bytes_read])
        )
        .transpose(),
)
```

Aux methods specialization

Using the lending async iterator

- Another helper method

What about specialized impl?

```
fn for_each<F>(
    self,
    f: F
) -> ForEach<Self, F>
```

Issues with specialized impl

- Now it’s not possible, because ForEach is returned
- It is possible to use an async fn
- It is possible to return an impl Future

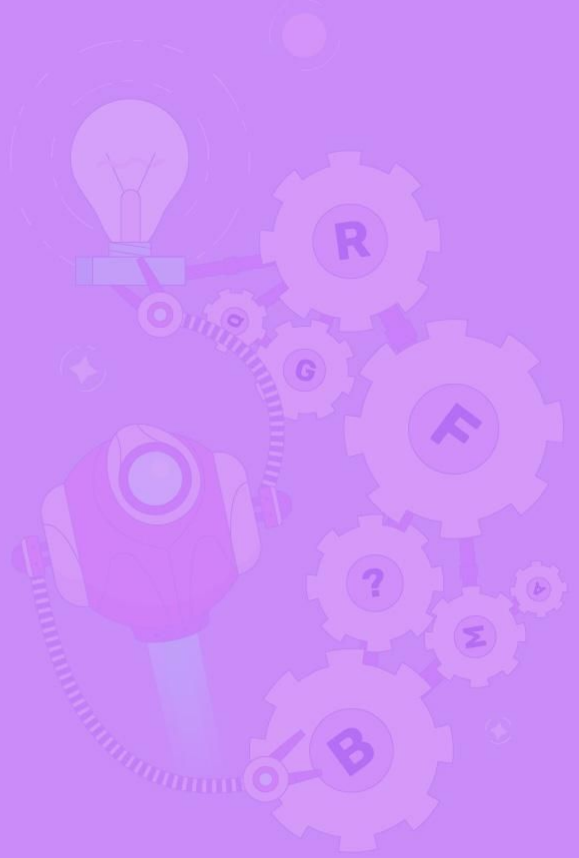
Issues with specialized impl

- Straightforward impl, rely on std::future::poll_fn (let's avoid using LendingAsyncIterator::next)

…but…

- You cannot store the actual future in stable, "type alias impl trait" (TAIT) is needed
- You cannot use bounds based on return type in stable, "return type notation" (RTN) is needed (unless an associated type is used, but "impl traits in associated type" is generally needed anyway)
- Sometimes it is not easy to create a specialization (see Filter)

The next method



Another helper function: next

```rust
fn next(&mut self) -> Next<'_, Self> {
    Next(self)
}

#[derive(Debug)]
pub struct Next<'a, T: ?Sized>(&'a mut T);
```

Another helper function: next

```
impl<'a, T> Future for Next<'a, T>
where
    T: ?Sized
        + LendingAsyncIterator
        + Unpin,
{
    type Output = Option<T::Item<'a>>;
    /* fn poll */
}
```

Another helper function: next

```
impl<'a, T> Future for Next<'a, T>
where
    T: ?Sized
        + LendingAsyncIterator
        + Unpin,
{
    type Output = Option<T::Item<'a>>;
    /* fn poll */
}
```

Another helper function: next

```rust
fn poll<'self>(
    self: Pin<&'self mut Self>,
    cx: &mut task::Context<'_>,
) -> Poll<Self::Output> {
    let inner = Pin::into_inner(self);
    Pin::new(&mut *inner.0).poll_next(cx)
}
```

Another helper function: next

```
fn poll<'self>(
    self: Pin<&'self mut Self>,
    cx: &mut task::Context<'_>,
) -> Poll<Self::Output> {
    let inner = Pin::into_inner(self);
    Pin::new(&mut *inner.0).poll_next(cx)
}
```

Another helper function: next

```
fn poll<'self>(
    self: Pin<&'self mut Self>,
    cx: &mut task::Context<'_>,
) -> Poll<Self::Output> {
    let inner = Pin::into_inner(self);
    Pin::new(&mut *inner.0).poll_next(cx)
}
```

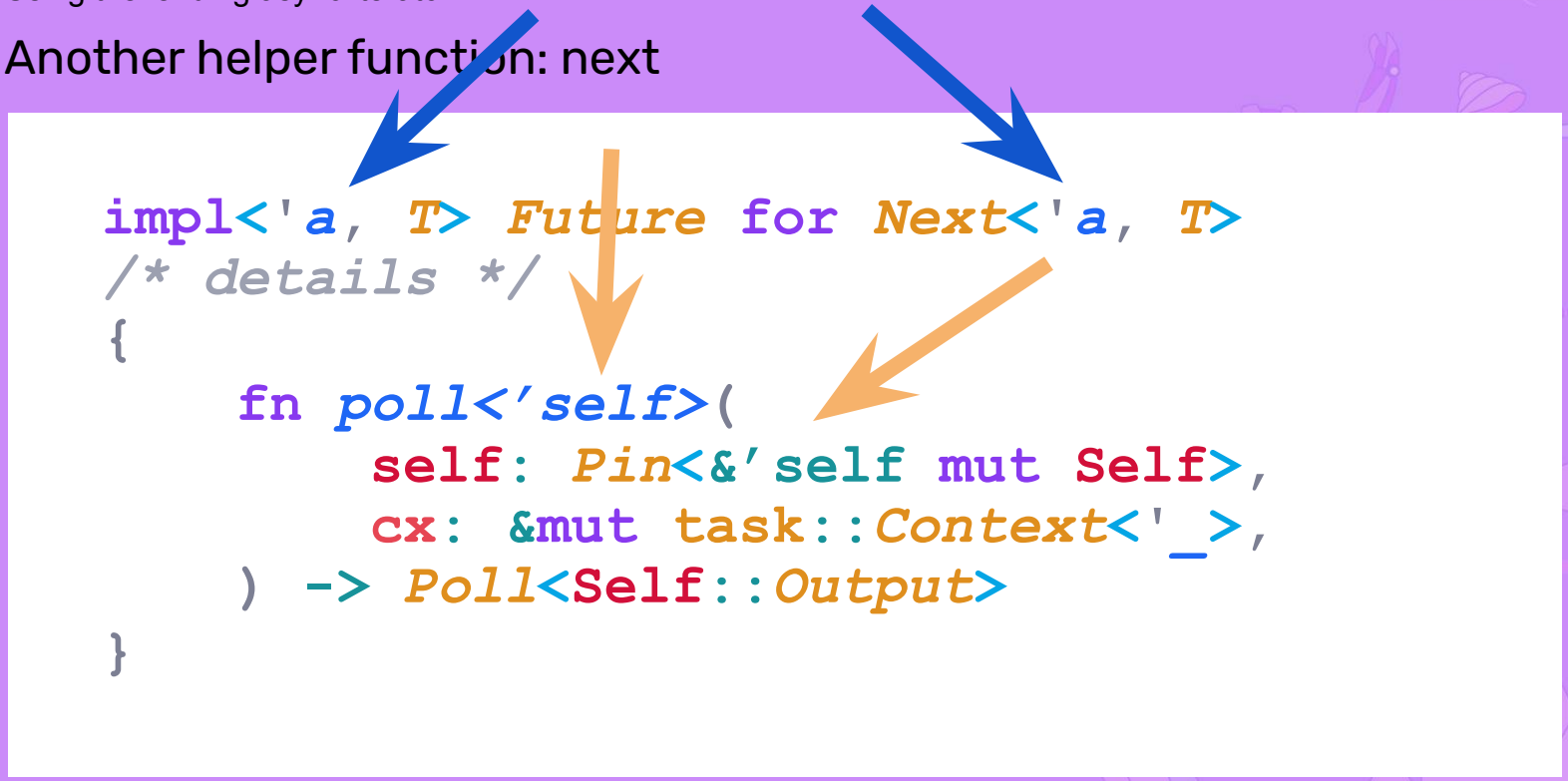
Another helper function: next

```
fn poll<'self>(
    self: Pin<&'self mut Self>,
    cx: &mut task::Context<'_>,
) -> Poll<Self::Output> {
    let inner = Pin::into_inner(self);
    Pin::new(&mut *inner.0).poll_next(cx)
}
```

- Another helper function: next 😭

```
method was supposed to
return data with lifetime
`'a` but it is returning
data with lifetime `'self`
```

- Another helper function: next



```
impl<'a, T> Future for Next<'a, T>
/* details */
{
    fn poll<'self>(
        self: Pin<&'self mut Self>,
        cx: &mut task::Context<'_>,
    ) -> Poll<Self::Output>
}
```

Another helper function

```
method was supposed to
return data with lifetime
`'a` but it is returning
data with lifetime `'self`
```

The `next` function needs to return a `LendingFuture`

```
trait LendingFuture {
    type Output<'a>
    where
        Self: 'a;

    fn poll<'a>(
        self: Pin<&'a mut Self>,
        cx: &mut task::Context,
    ) -> Poll<Self::Output<'a>>;
}
```