Transforming `Streams`

Advanced stream processing in Rust

Willem Vanhulle

EuroRust 2025

1980-01-01

1.1. Me

Motivation

Processing data from moving vehicles

1. Vehicle generates multiple data streams
2. All streams converge to control system

1.1. Me

Lives in Ghent, Belgium:

- Studied mathematics, physics and computer science
- Biotech automation (fermentation)
- Distributed systems (trains)

Motivation

Processing data from moving vehicles

1. Vehicle generates multiple data streams
2. All streams converge to control system

1.1. Me

Lives in Ghent, Belgium:

- Studied mathematics, physics and computer science
- Biotech automation (fermentation)
- Distributed systems (trains)

Latest projects (github.com/wvhulle):

- SysGhent.be: social network for systems programmers in Ghent (Belgium)
- Clone-stream: lazy stream cloning library for Rust

Motivation

Processing data from moving vehicles

1. Vehicle generates multiple data streams
2. All streams converge to control system

1.2. Kinds of streams

Physical streams

Electronic signals

GPIO interrupts • UART frames • Network packets

1.2. Kinds of streams

Requires an async runtime
(see ‘leaf future’ by *Carl Fredrik Samson*) →

Leaf streams

OS/kernel constraints

`tokio::fs::File` • `TcpListener` • `UnixStream` • `Interval`

↑ OS abstraction

Physical streams

Electronic signals

GPIO interrupts • UART frames • Network packets

1.2. Kinds of streams

In this presentation →

Stream operators

Pure software transformations

`map()` • **`double()`** • **`fork()`** • **`latency()`** • **`hysteresis()`**

Stream operators

Requires an async runtime
(see 'leaf future' by *Carl Fredrik Samson*) →

Leaf streams

OS/kernel constraints

`tokio::fs::File` • `TcpListener` • `UnixStream` • `Interval`

OS abstraction

Physical streams

Electronic signals

GPIO interrupts • UART frames • Network packets

Operators

Streams

Data

1.2. Kinds of streams

In this presentation →

Stream operators

Pure software transformations

`map()` • **`double()`** • **`fork()`** • **`latency()`** • **`hysteresis()`**

↑ Stream operators

Requires an async runtime →

(see ‘leaf future’ by *Carl Fredrik Samson*)

Leaf streams

OS/kernel constraints

`tokio::fs::File` • `TcpListener` • `UnixStream` • `Interval`

↑ OS abstraction

Physical streams

Electronic signals

GPIO interrupts • UART frames • Network packets

Operators

Streams

Data

Hardware signals are abstracted by the OS

Software operators transform the streams

1.3. `Streams` in Rust are not new

1978

Communicating Sequential Processes
Tony Hoare formalizes concurrent dataflow
CACM 21(8):666-677

1973

Unix Pipes
Douglas McIlroy creates `|` operator
Bell Labs, Unix v3-v4

1960s

Dataflow Programming
Hardware-level stream processing
Early dataflow architectures

1.3. Streams in Rust are not new

1997	**Functional Reactive Programming** Conal Elliott & Paul Hudak (Haskell) *ICFP '97, pp. 263-273*
1978	**Communicating Sequential Processes** Tony Hoare formalizes concurrent dataflow *CACM 21(8):666-677*
1973	**Unix Pipes** Douglas McIlroy creates \`|\` operator *Bell Labs, Unix v3-v4*
1960s	**Dataflow Programming** Hardware-level stream processing *Early dataflow architectures*

1.3. Streams in Rust are not new

2009	**Microsoft Reactive Extensions** ReactiveX brings streams to mainstream *Erik Meijer, Microsoft*
1997	**Functional Reactive Programming** Conal Elliott & Paul Hudak (Haskell) *ICFP '97, pp. 263-273*
1978	**Communicating Sequential Processes** Tony Hoare formalizes concurrent dataflow *CACM 21(8):666-677*
1973	**Unix Pipes** Douglas McIlroy creates \`|\` operator *Bell Labs, Unix v3-v4*
1960s	**Dataflow Programming** Hardware-level stream processing *Early dataflow architectures*

1.3. Streams in Rust are not new

2019
async/await stabilized in Rust
Stable async streams in std
RFC 2394, Rust 1.39.0

2009
Microsoft Reactive Extensions
ReactiveX brings streams to mainstream
Erik Meijer, Microsoft

1997
Functional Reactive Programming
Conal Elliott & Paul Hudak (Haskell)
ICFP '97, pp. 263-273

1978
Communicating Sequential Processes
Tony Hoare formalizes concurrent dataflow
CACM 21(8):666-677

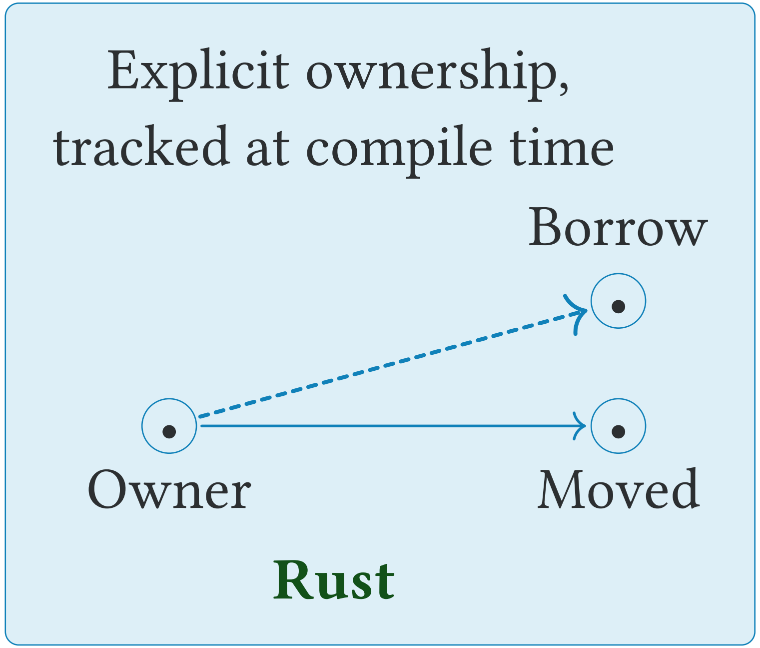
1973
Unix Pipes
Douglas McIlroy creates `|` operator
Bell Labs, Unix v3-v4

1960s
Dataflow Programming
Hardware-level stream processing
Early dataflow architectures

1.4. Why does Rust need special treatment?

Stream operators must wrap and own their input by value

vs



1.5. Process TCP connections and collect long messages

Rust

```rust
let mut results = Vec::new(); let mut count = 0;
while let Some(connection) = tcp_stream.next().await {
    match connection {
        Ok(stream) if should_process(&stream) => {
            match process_stream(stream).await {
                Ok(msg) if msg.len() > 10 => {
                    results.push(msg);
                    count += 1;
                    if count >= 5 { break; }
                }
                Ok(_) => continue,
                Err(_) => continue,
            }
        }
        Ok(_) => continue,
        Err(_) => continue,
    }
}
```

Problems:

- Deeply nested
- Hard to read
- Cannot test pieces independently

1.6. `Stream` operators: declarative & composable

Same logic with stream operators:

```rust
let results: Vec<String> = tcp_stream
        .filter_map(|conn| ready(conn.ok()))
        .filter(|stream| ready(should_process(stream)))
        .then(|stream| process_stream(stream))
        .filter_map(|result| ready(result.ok()))
        .filter(|msg| ready(msg.len() > 10))
        .take(5)
        .collect()
        .await;
```

Benefits:

- Each operation is isolated
- Testable
- Reusable

"Programs must be written **for people to read**"

2.1. Moving from `Iterator` to `Stream`

✓ Always returns immediately

⚠️ May be Pending

✓ Hides polling complexity

2.1. Moving from `Iterator` to `Stream`

✓ Always returns immediately

`next()` ⟶ `Some(3)`

`next()` ⟶ `Some(1)`

`next()` ⟶ `None`

`next()` ⟶ `Some(2)`

Iterator (sync)

⚠️ May be Pending

✓ Hides polling complexity

2.1. Moving from `Iterator` to `Stream`

✓ Always returns immediately

⚠️ May be Pending

✓ Hides polling complexity

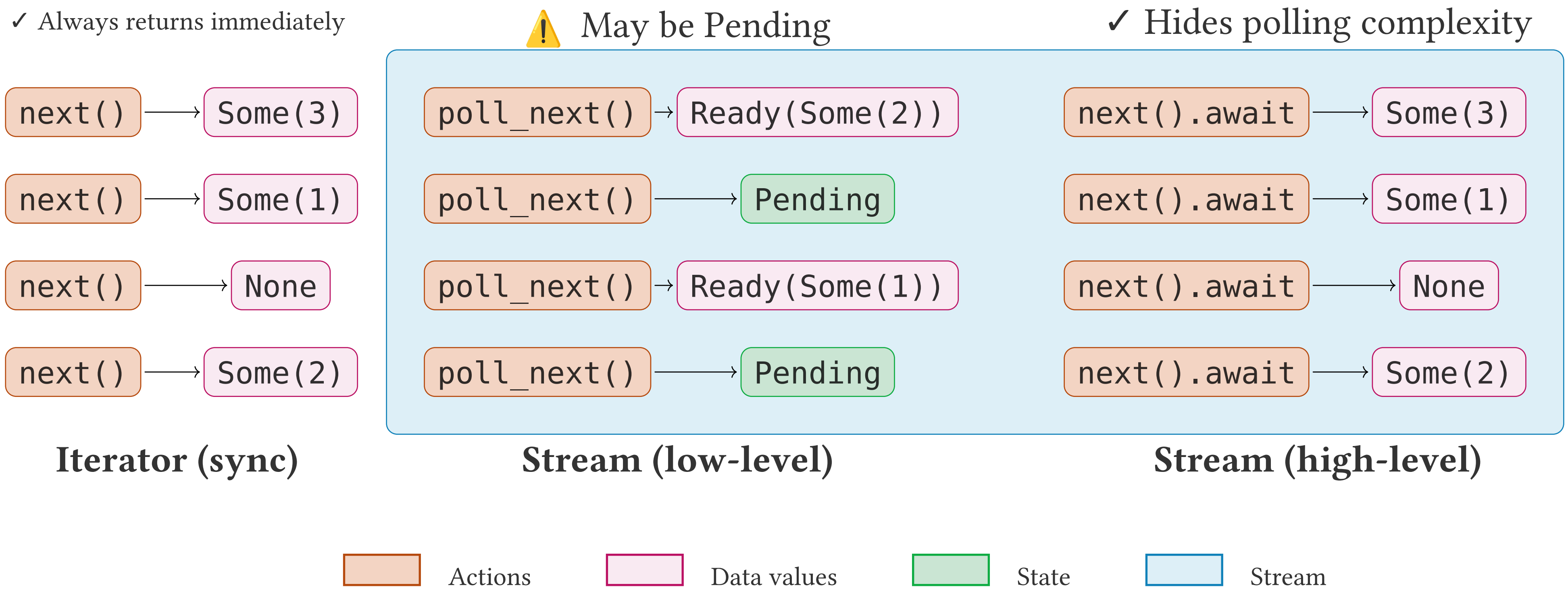
`next()` → `Some(3)`
`next()` → `Some(1)`
`next()` → `None`
`next()` → `Some(2)`

Iterator (sync)

`poll_next()` → `Ready(Some(2))`
`poll_next()` → `Pending`
`poll_next()` → `Ready(Some(1))`
`poll_next()` → `Pending`

Stream (low-level)

2.1. Moving from `Iterator` to `Stream`



2.2. The Stream trait: async iterator

Like `Future`, but yields **multiple items** over time when polled:

```rust
trait Stream {
      type Item;

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

The `Poll<Option<Item>>` return type:

- `Poll::Pending` - not ready yet, try again later
- `Poll::Ready(Some(item))` - here's the next item
- `Poll::Ready(None)` - stream is exhausted (no more items **right now**)

2.3. Possible inconsistency

```rust
trait Stream {
    type Item;

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

Warning

What about Rust rule `self` needs to be `Deref<Target=Self>`?

`Pin<&mut Self>` only implements `Deref<Target=Self>` for `Self: Unpin`.

Problem? No, `Pin` is an exception in the compiler.

2.4. The meaning of `Ready(None)`

Regular Stream

"No items **right now**"

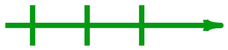
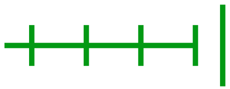
(*Stream might yield more later*)

Fused Stream

"No items **ever again**"

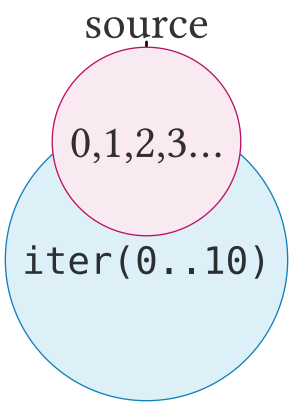
(*Stream is permanently done*)

2.5. 'Fusing' Streams and Futures

	`Future`	**`Stream`**	**Meaning**
Regular			May continue
Fused	**`FusedFuture`**	**`FusedStream`**	`is_terminated()` method
Fused			Done permanently
Fused value	`Pending`	`Ready(None)`	Final value

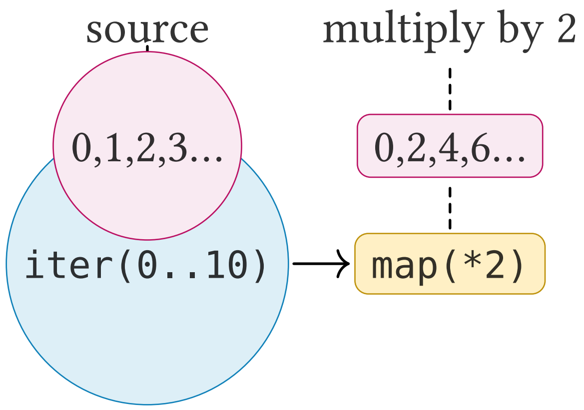
3.1. Pipelines with `futures::StreamExt`

All basic stream operators are in `futures::StreamExt`



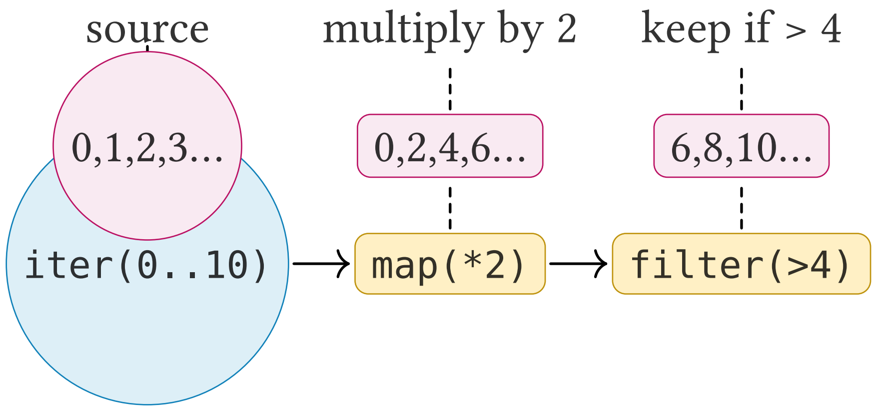
3.1. Pipelines with `futures::StreamExt`

All basic stream operators are in `futures::StreamExt`



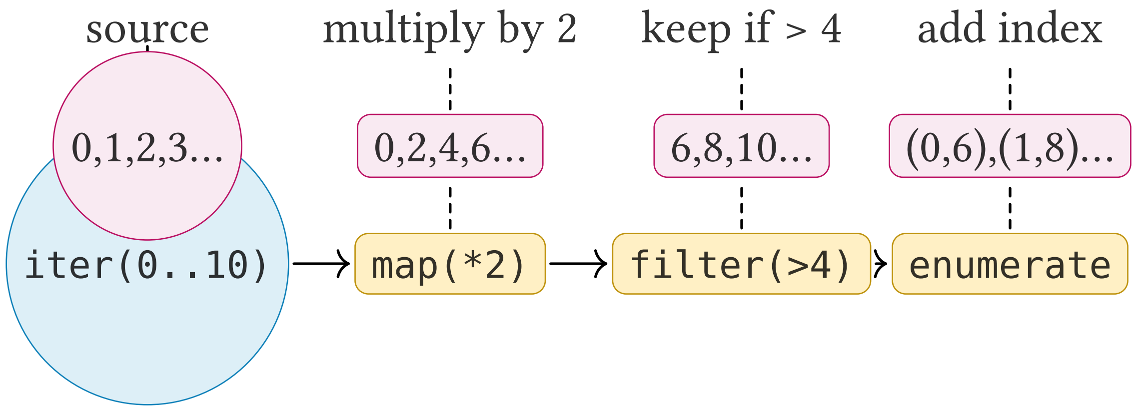
3.1. Pipelines with `futures::StreamExt`

All basic stream operators are in `futures::StreamExt`



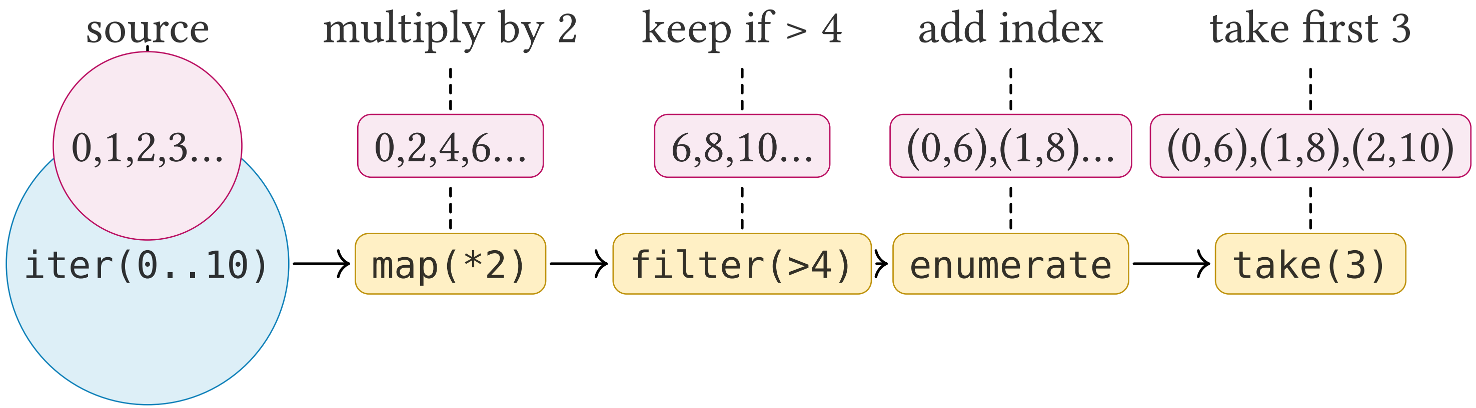
3.1. Pipelines with `futures::StreamExt`

All basic stream operators are in `futures::StreamExt`



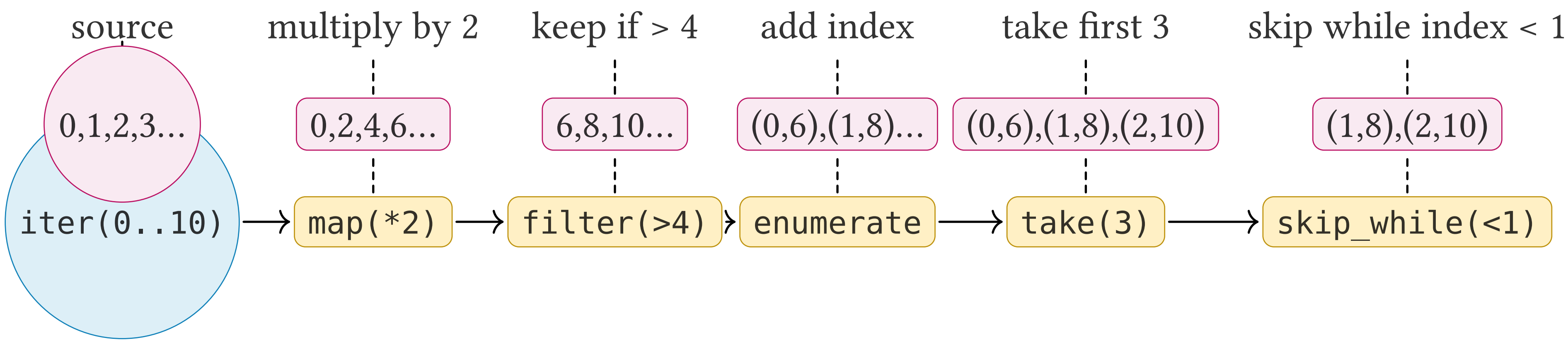
3.1. Pipelines with `futures::StreamExt`

All basic stream operators are in `futures::StreamExt`



3.1. Pipelines with `futures::StreamExt`

All basic stream operators are in `futures::StreamExt`




```rust
stream::iter(0..10)
    .map(|x| x * 2)
    .filter(|&x| ready(x > 4))
    .enumerate()
    .take(3)
    .skip_while(|&(i, _)| i < 1)
```

3.2. The handy `std::future::ready` function

The `futures::StreamExt::filter` expects an **async closure** (or closure returning `Future`):

Option 1: Async block (not `Unpin`!)

```rust
stream.filter(|&x| async move {
    x % 2 == 0
})
```

rust

Option 2: Async closure (not `Unpin`!)

```rust
stream.filter(async |&x| x % 2 == 0)
```

rust

Option 3 (recommended): Wrap sync output with `std::future::ready()`

```rust
stream.filter(|&x| ready(x % 2 == 0))
```

rust

- `ready(value)` creates a `Future` that immediately resolves to `value`.
- `ready(value)` is `Unpin`

The ready trick

`ready` keeps pipelines `Unpin`: *easier to work with*

3.3. Flatten a finite collection of `Streams`

A finite collection of `Streams` = `IntoIterator<Item: Stream>`

```rust
let streams = vec![
        stream::iter(1..=3),
        stream::iter(4..=6),
        stream::iter(7..=9),
];

let merged = stream::select_all(streams);
```

1. Creates a `FuturesUnordered` of the streams
2. Polls all streams concurrently
3. Yields items as they arrive

3.4. Flattening an infinite stream

Beware!: `flatten()` on a stream of infinite streams will never complete!

```rust
let infinite_streams = stream::unfold(0, |id| async move {
    Some((stream::iter(id..), id + 1))
});
let flat = infinite_streams.flatten();
```

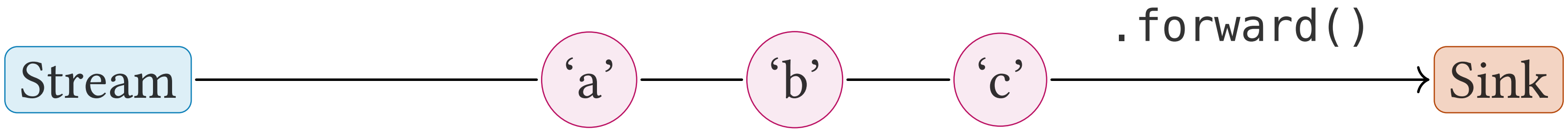
Instead, **buffer streams** concurrently with `flatten_unordered()`.

```rust
let requests = stream::unfold(0, |id| async move {
    Some((fetch_stream(format!("/api/data/{}", id)), id + 1))
});
let flat = requests.flatten_unordered(Some(10));
```

3.5. More `Stream` features to explore

Many more advanced topics await:

- **Boolean operations**: `any`, `all`
- **Async operations**: `then`
- **`Sinks`**: The write-side counterpart to `Streams`

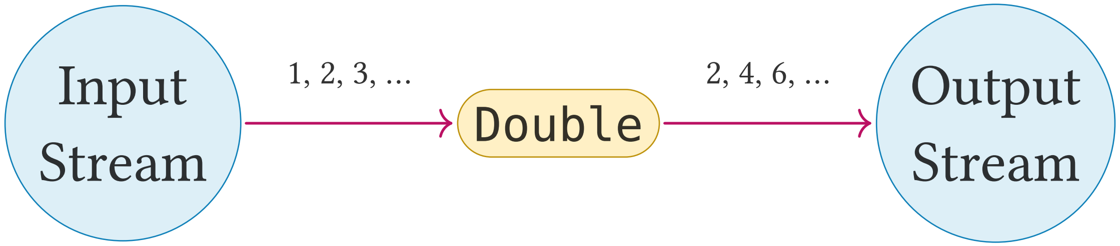


Read side

Write side

4.1. Doubling stream operator

Very simple `Stream` operator that **doubles every item** in an input stream:



Input stream **needs to yield integers**.

4.1. Doubling stream operator

Step 1: Define a struct that wraps the input stream

rust

```rust
struct Double<InSt> {
    in_stream: InSt,
}
```

- Generic over stream type (works with any backend)
- Stores input stream by value

4.1. Doubling stream operator

Step 2: Implement `Stream` trait with bounds

```rust
impl<InSt> Stream for Double<InSt>
where
    InSt: Stream<Item = i32>
{
    type Item = i32;

    fn poll_next(self: Pin<&mut Self>, cx: &mut Context<'_>)
        -> Poll<Option<Self::Item>> {
        // ... implementation goes here
    }
}
```

4.2. Naive implementation of poll_next

Focus on the implementation of the `poll_next` method

(Remember that `Self = Double<InSt>` with field `in_stream: InSt`):

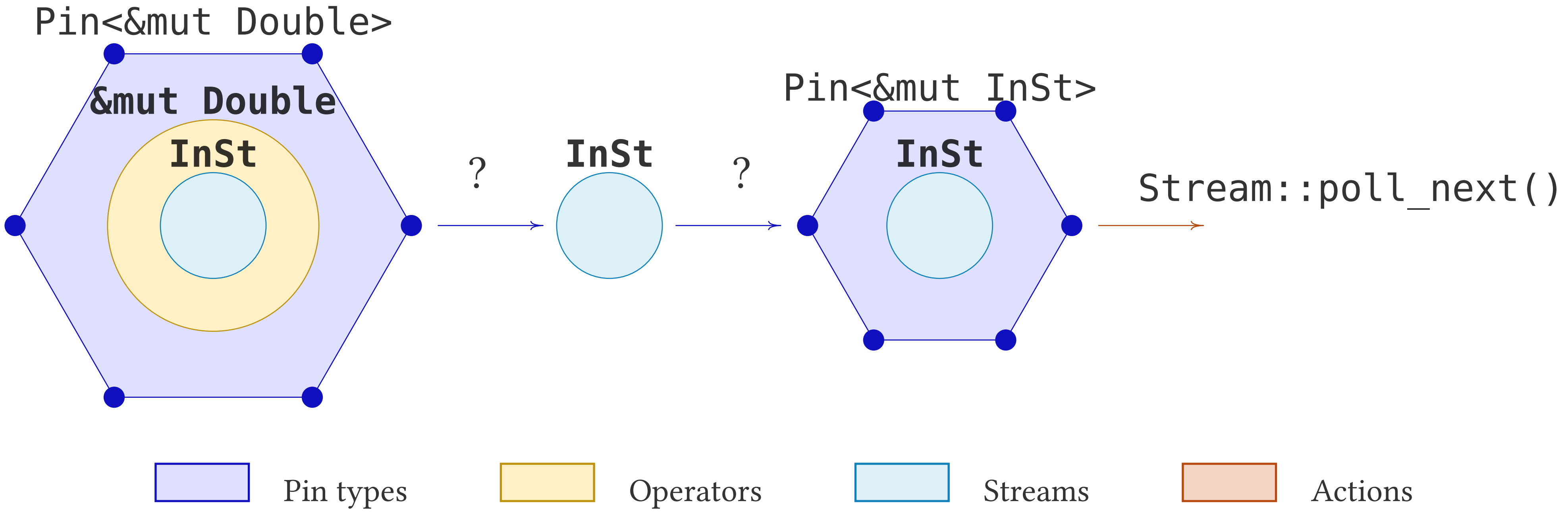
```rust
fn poll_next(self: Pin<&mut Self>, cx: &mut Context<'_>)
    -> Poll<Option<Self::Item>> {
        // Cannot access self.in_stream!
        Pin::new(&mut self.in_stream) // Not possible!
            .poll_next(cx)
            .map(|x| x * 2)
}
```

`Pin<&mut Self>` **blocks access to `self.in_stream`** (when `Self: !Unpin`)!

4.2. Naive implementation of `poll_next`

We have `Pin<&mut Double>`

How can we obtain `Pin<&mut InSt>` to call `poll_next()`?



4.2. Naive implementation of `poll_next`

Can we use `Pin::get_mut()` to unwrap and re-wrap?

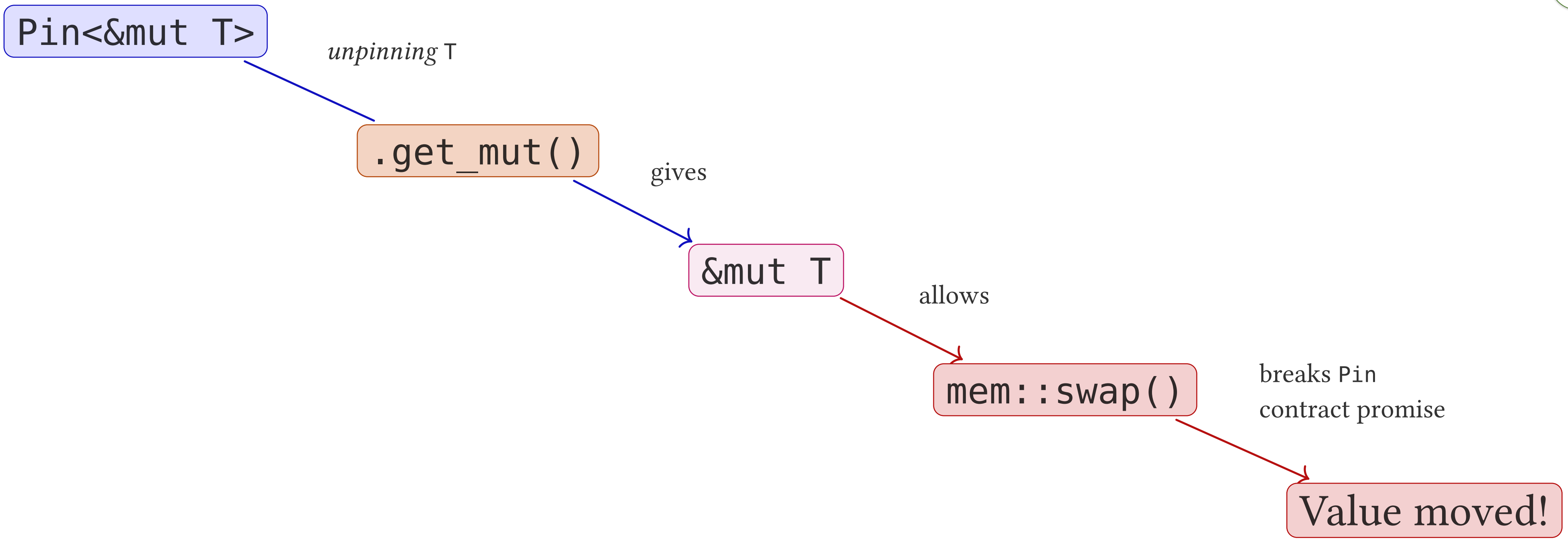
```rust
impl<InSt> Stream for Double<InSt> where InSt: Stream<Item = i32> {

    type Item = InSt::Item;

    fn poll_next(self: Pin<&mut Self>, cx: &mut Context<'_>)
          -> Poll<Option<Self::Item>> {
        let this = self.get_mut();  // Error!
        let pinned_in = Pin::new(&mut this.in_stream);
        pinned_in.poll_next(cx).map(|p| p.map(|x| x * 2))
    }
}
```

Problem: `Pin::get_mut()` requires `Double<InSt>: Unpin`

But ***`Double<InSt>` is `!Unpin`*** *when `InSt: !Unpin`*!

4.3. Why does `Pin::get_mut()` require `Unpin`?

`Pin<P>` makes a promise: **the pointee will never move again**.



Solution

Only allow `get_mut()` when `T: Unpin` (moving is safe).

4.4. `Unpin` types can be safely unpinned

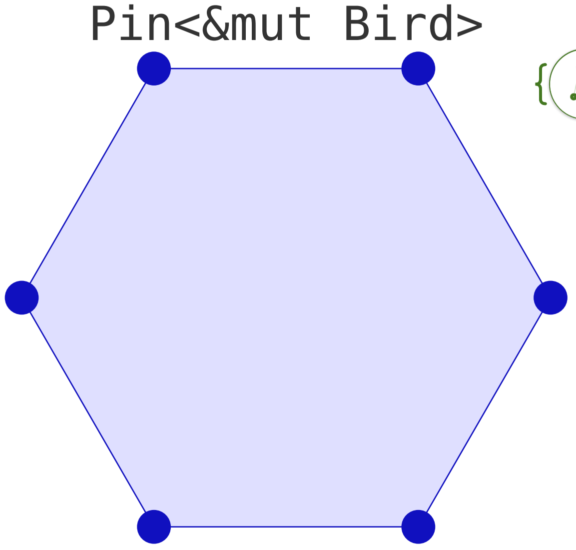
`Unpin` Bird

Safe to move

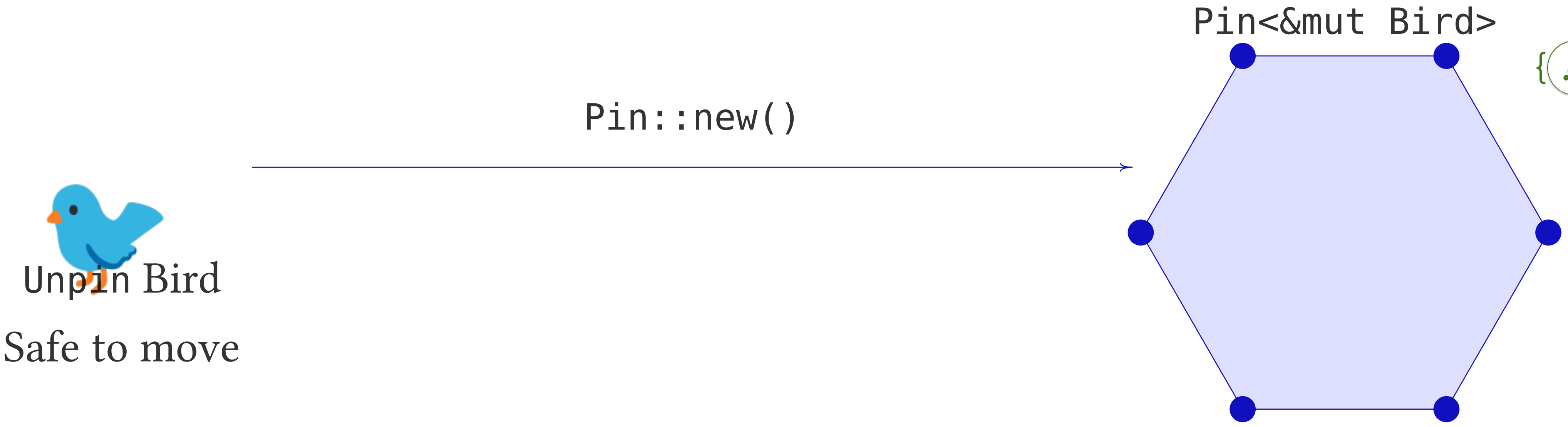
4.4. `Unpin` types can be safely unpinned

`Unpin` Bird

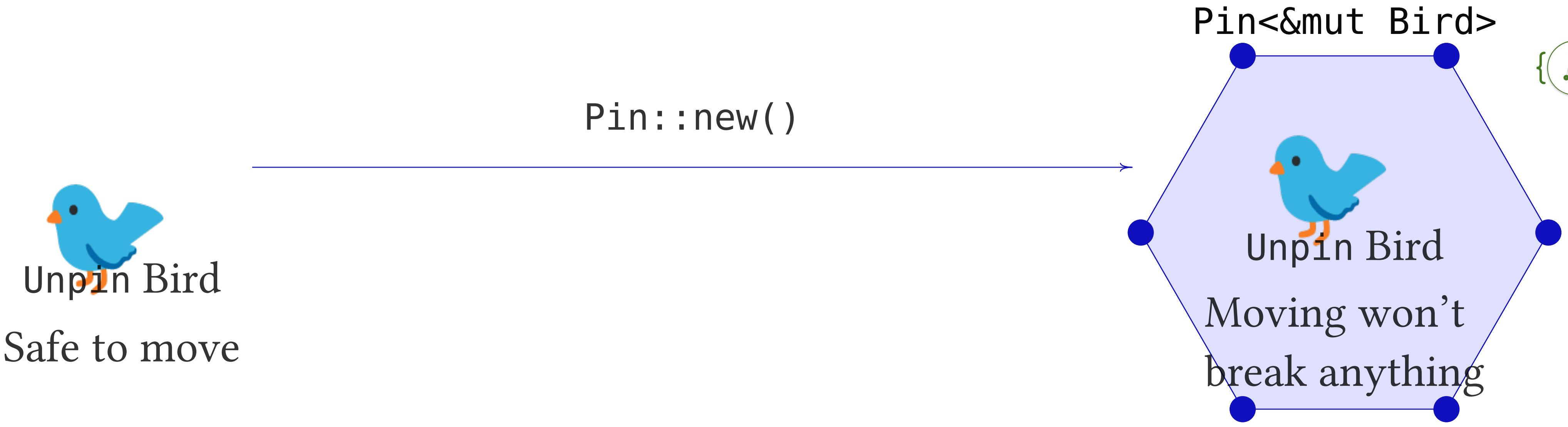
Safe to move



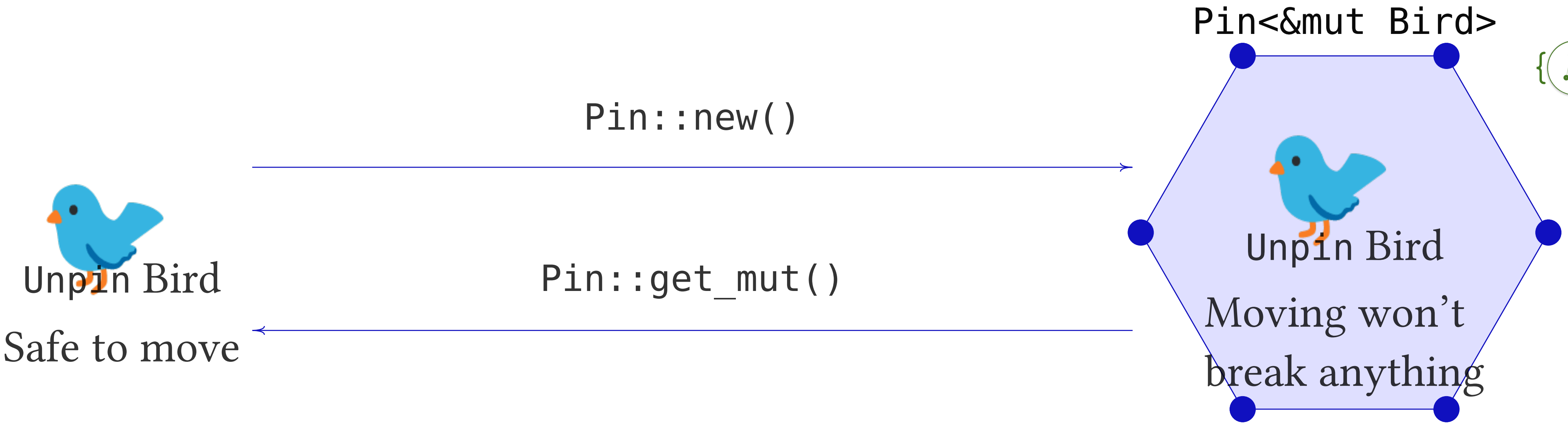
4.4. `Unpin` types can be safely unpinned



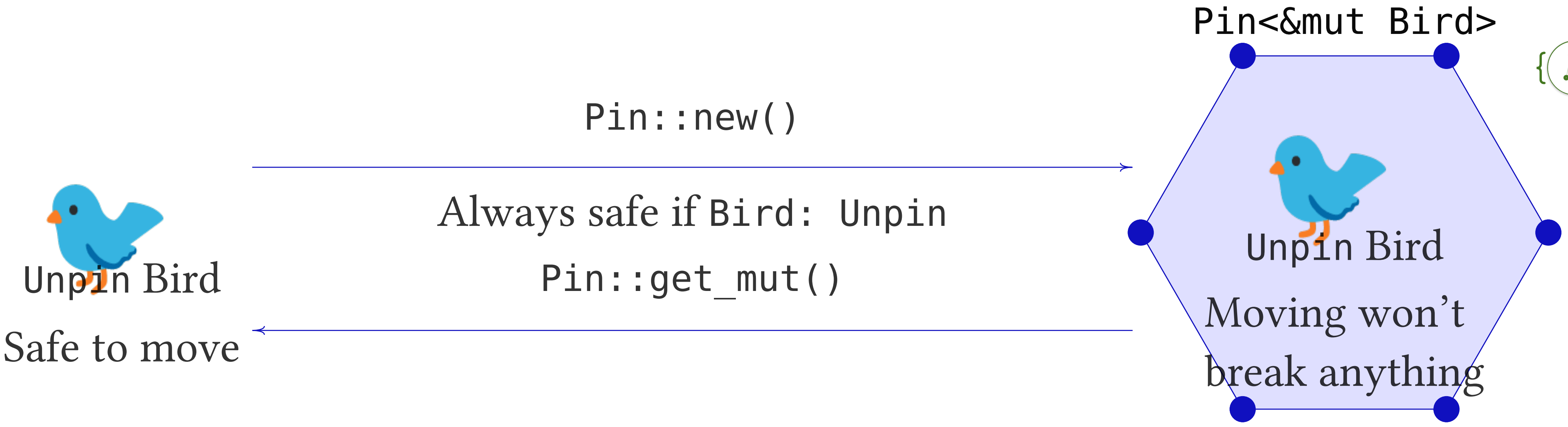
4.4. `Unpin` types can be safely unpinned



4.4. `Unpin` types can be safely unpinned



4.4. `Unpin` types can be safely unpinned



If `T: Unpin`, then `Pin::get_mut()` is safe because moving `T` doesn't cause UB.

4.4. `Unpin` types can be safely unpinned

Examples of `Unpin` types:

- `i32`, `String`, `Vec<T>` - all primitive and standard types
- `Box<T>` - pointers are safe to move
- `&T`, `&mut T` - references are safe to move

Why safe?

These types don't have self-referential pointers. Moving them in memory doesn't invalidate any internal references.

Info

Almost all types are `Unpin` by default!

4.5. `!Unpin` types cannot be unpinned

`!Unpin` Tiger

Dangerous to move

~~`Pin::get_mut()` gives `&mut T`~~

Would break pin promise!

`Pin<&mut Tiger>`

`!Unpin` Tiger

Moving would cause UB

4.5. `!Unpin` types cannot be unpinned

Examples of `!Unpin` types:

- `PhantomPinned` - explicitly opts out of `Unpin`
- Most `Future` types (self-ref. state machines)
- Types with self-referential pointers
- `Double<InSt>` where `InSt: !Unpin`

Why unsafe?

These types may contain pointers to their own fields. Moving them in memory would invalidate those internal pointers, causing use-after-free.

Info

`!Unpin` is rare and usually intentional for async/self-referential types.

4.6. Following compiler hints

The compiler error suggests adding `InSt: Unpin`:

```rust
impl<InSt> Stream for Double<InSt> where InSt: Stream<Item = i32> + Unpin {
  type Item = InSt::Item;

  fn poll_next(self: Pin<&mut Self>, cx: &mut Context<'_>) ->  Poll<Option<Self::Item>> {
    // `this` = a conventional name for `get_mut` output
    let mut this = self.get_mut();
    let pinned_in = Pin::new(&mut this.in_stream);
    pinned_in
      .poll_next(cx)
      .map(|p| p.map(|x| x * 2))
  }
}
```

4.6. Following compiler hints

The compiler error suggests adding `InSt: Unpin`:

```rust
impl<InSt> Stream for Double<InSt> where InSt: Stream<Item = i32> + Unpin {
  type Item = InSt::Item;

  fn poll_next(self: Pin<&mut Self>, cx: &mut Context<'_>) ->  Poll<Option<Self::Item>> {
    // `this` = a conventional name for `get_mut` output
    let mut this = self.get_mut();
    let pinned_in = Pin::new(&mut this.in_stream);
    pinned_in
      .poll_next(cx)
      .map(|p| p.map(|x| x * 2))
  }
}
```

Warning

This is a common, misleading compiler hint and **not the right solution**!

4.6. Following compiler hints

Instead of mindlessly following the compiler suggestion:

Info

Accept that `!Unpin` things are a fact of life and ask your users to pin stream operators (or futures and other raw `!Unpin` types):

- On the stack with the `pin!` macro
- On the heap with `Box::new()`

4.6. Following compiler hints

Instead of mindlessly following the compiler suggestion:

Info

Accept that `!Unpin` things are a fact of life and ask your users to pin stream operators (or futures and other raw `!Unpin` types):

- On the stack with the `pin!` macro
- On the heap with `Box::new()`

Instead of forcing customers of our API to know what `Unpin` means, I decided to "fix" the problem upstream and pin on the heap.

4.6. Following compiler hints

Instead of mindlessly following the compiler suggestion:

Info

Accept that `!Unpin` things are a fact of life and ask your users to pin stream operators (or futures and other raw `!Unpin` types):

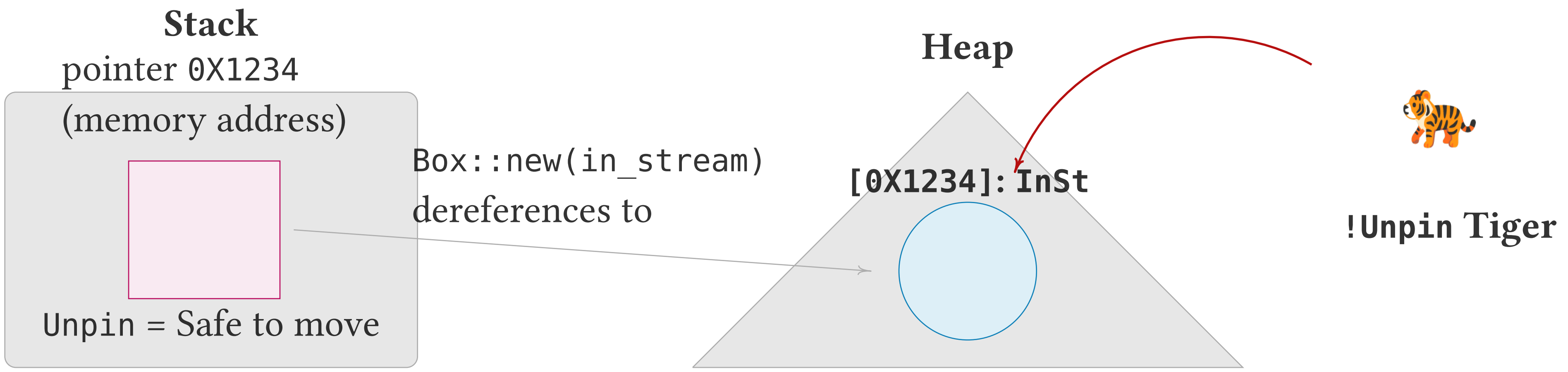
- On the stack with the `pin!` macro
- On the heap with `Box::new()`

Instead of forcing customers of our API to know what `Unpin` means, I decided to "fix" the problem upstream and pin on the heap.

Warning

Pinning the original stream on the heap is not a **real** / idiomatic Rust solution!
(-0-30% runtime performance)

4.7. Turning `!Unpin` into `Unpin` with boxing



Nice to have:

1. `Box::new(tiger)` produces just a pointer on the stack
   - Moving pointers is always safe
   - Therefore: **`Box<Tiger>: Unpin`**
2. Box dereferences to its contents
   - **`Box<X>: Deref<Target = X>`**

Problem: Need `Pin<&mut InSt>`, but `Box<InSt>` requires `InSt: Unpin` to create it

Solution

Use `Pin<Box<InSt>>` to project from `Pin<&mut Double>` to `Pin<&mut InSt>` via `Pin::as_mut()`

4.7. Turning `!Unpin` into `Unpin` with boxing

Change the struct definition to store `Pin<Box<InSt>>`:

```rust
struct Double<InSt> { in_stream: Pin<Box<InSt>>, }
```

Why this works:

- `Box<InSt>` is always `Unpin` (pointers are safe to move)
- `Pin<Box<InSt>>` can hold `!Unpin` streams safely on the heap

4.7. Turning `!Unpin` into `Unpin` with boxing

Change the struct definition to store `Pin<Box<InSt>>`:

```rust
struct Double<InSt> { in_stream: Pin<Box<InSt>>, }
```

Why this works:

- `Box<InSt>` is always `Unpin` (pointers are safe to move)
- `Pin<Box<InSt>>` can hold `!Unpin` streams safely on the heap

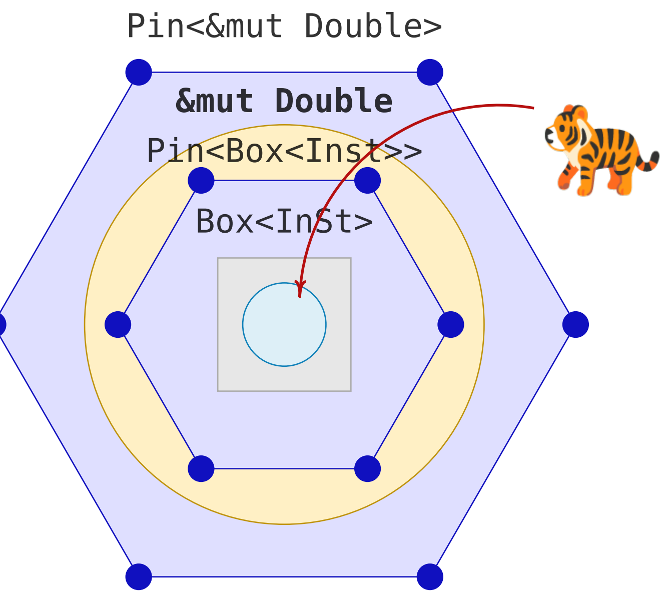
Projection in `poll_next`:

```rust
fn poll_next(self: Pin<&mut Self>, cx: &mut Context<'_>)
    -> Poll<Option<Self::Item>> {
    let this = self.get_mut();  // Safe: Double is Unpin now
    this.in_stream.as_mut()      // Project to Pin<&mut InSt>
        .poll_next(cx)
        .map(|opt| opt.map(|x| x * 2))
  }
```

This works **without requiring `InSt: Unpin`**!

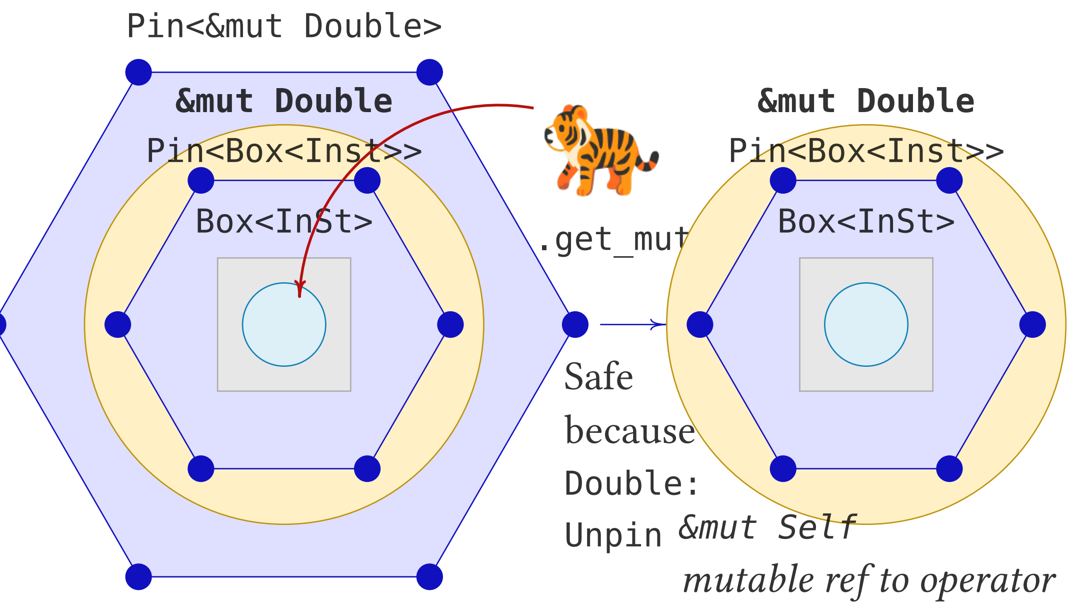
4.7. Turning `!Unpin` into `Unpin` with boxing

From `Pin<&mut Double>` to `Pin<&mut InSt>` in a few **safe steps**:



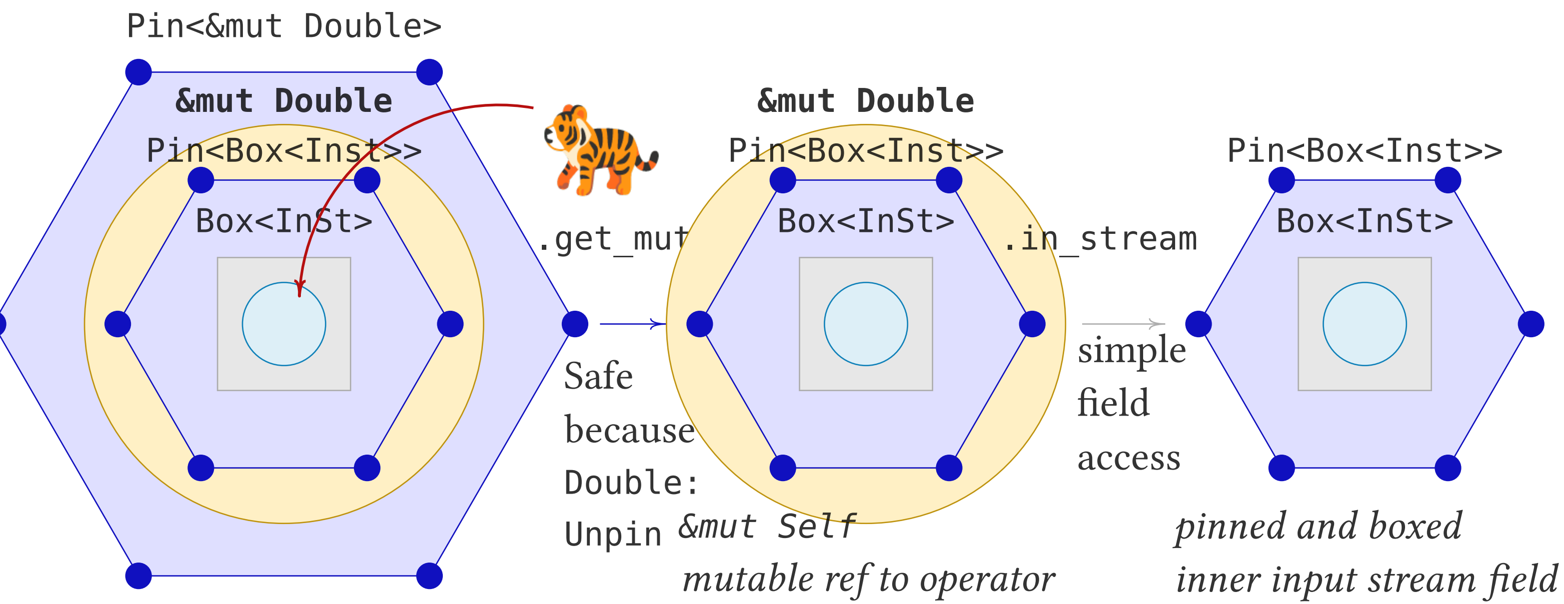
4.7. Turning `!Unpin` into `Unpin` with boxing

From `Pin<&mut Double>` to `Pin<&mut InSt>` in a few **safe steps**:



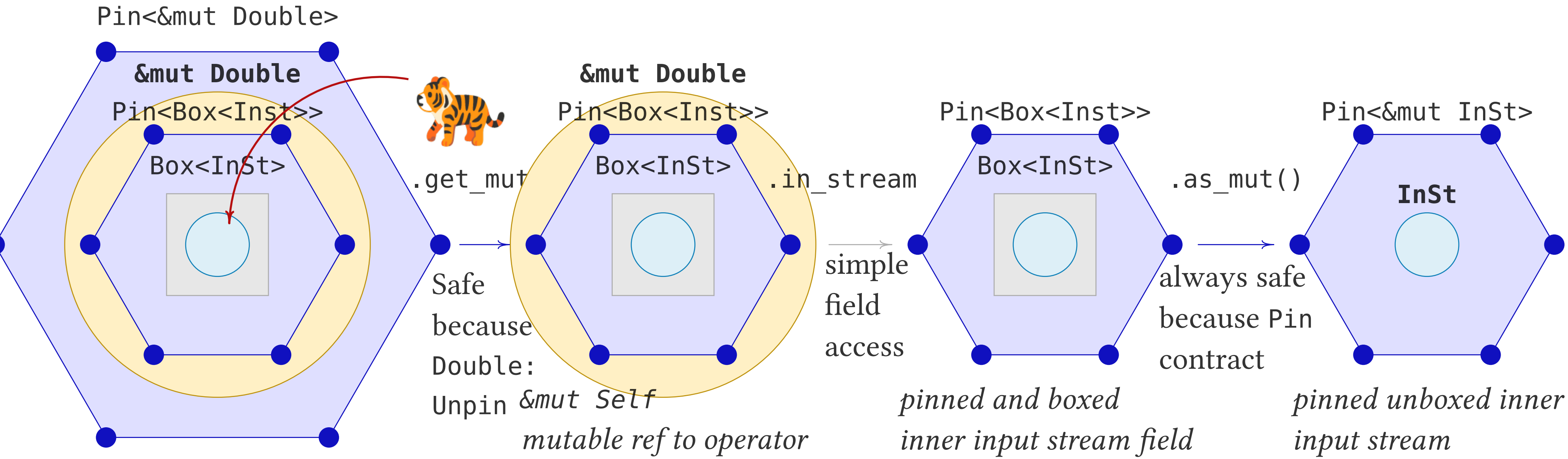
4.7. Turning `!Unpin` into `Unpin` with boxing

From `Pin<&mut Double>` to `Pin<&mut InSt>` in a few **safe steps**:



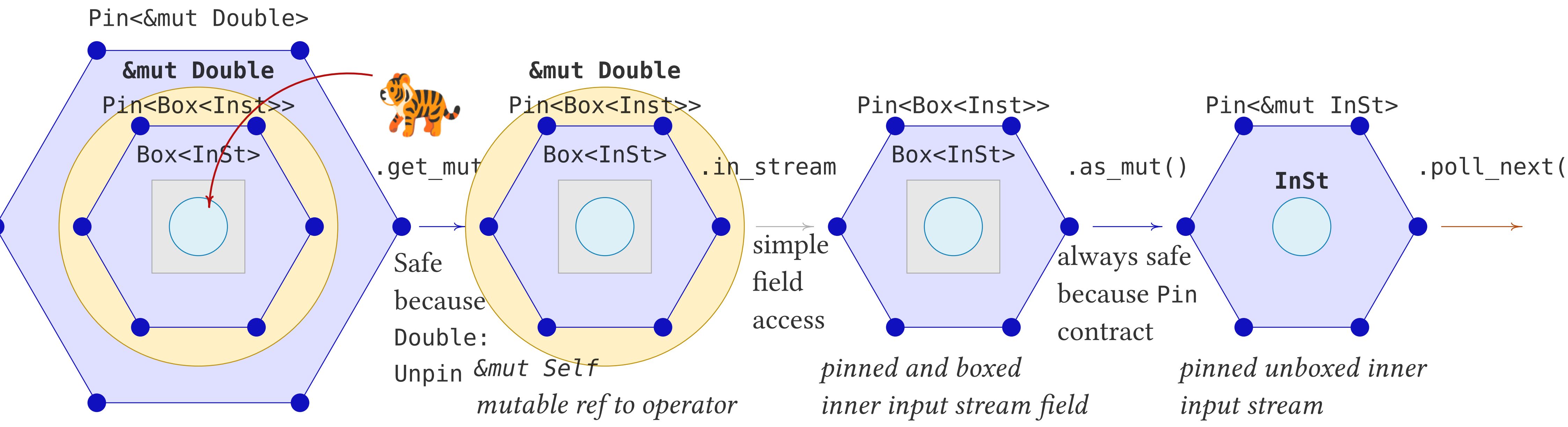
4.7. Turning `!Unpin` into `Unpin` with boxing

From `Pin<&mut Double>` to `Pin<&mut InSt>` in a few **safe steps**:



4.7. Turning `!Unpin` into `Unpin` with boxing

From `Pin<&mut Double>` to `Pin<&mut InSt>` in a few **safe steps**:



4.7. Turning !Unpin into Unpin with boxing

We can call `Pin::get_mut()` to get `&mut Double<InSt>` safely from `Pin<&mut Double<InSt>>`

```rust
impl<InSt> Stream for Double<InSt>
where InSt: Stream<Item = i32>
{
    fn poll_next(self: Pin<&mut Self>, cx: &mut Context<'_>)
        -> Poll<Option<Self::Item>>
    {
        // We can project because `Self: Unpin`
        let this: &mut Double<InSt> = self.get_mut();
        this.in_stream.as_mut()
            .poll_next(cx)
            .map(|r| r.map(|x| x * 2))
    }
}
```

4.8. Review of approaches to `!Unpin` fields

Approach 1: Use `Box<_>`

```rust
struct Double<InSt> {
    in_stream: Pin<Box<InSt>>
}

impl<InSt> Stream for Double<InSt>
    where InSt: Stream
```

✓ Works with any `InSt`, also `!Unpin`

4.8. Review of approaches to `!Unpin` fields

Approach 1: Use `Box<_>`

```rust
struct Double<InSt> {
    in_stream: Pin<Box<InSt>>
}

impl<InSt> Stream for Double<InSt>
    where InSt: Stream
```

✓ Works with any `InSt`, also `!Unpin`

Approach 2: Require `Unpin`

```rust
struct Double<InSt> {
    in_stream: InSt
}

impl<InSt> Stream for Double<InSt>
    where InSt: Stream + Unpin
```

✗ Imposes `Unpin` constraint on users

Approach 3: Use `pin-project` crate

4.8. Review of approaches to `!Unpin` fields

Approach 3: Projection with `pin-project`

Do not impose `Unpin` constraint on input stream **and** avoid heap allocation with `Box`:

```rust
#[pin_project]
struct Double<InSt> {
    #[pin]
    in_stream: InSt,
}
impl<InSt: Stream> Stream for Double<InSt> {
    fn poll_next(self: Pin<&mut Self>, cx: &mut Context<'_>)
        -> Poll<Option<Self::Item>>
    {
        self.project().in_stream.poll_next(cx)
            .map(|r| r.map(|x| x * 2))
    }
}
```

Info

`pin-project` generates a safe projection method `project()`.

Warning

You don't have juggle with an `Unpin` constraint (**but your users have to!**)

4.9. Distributing your operator

Define a constructor and turn it into a method of an **extension trait**:

```rust
trait DoubleStream: Stream {
        fn double(self) -> Double<Self>
        where Self: Sized + Stream<Item = i32>,
        { Double::new(self) }
}
  // A blanket implementation should be provided by you!
  impl<S> DoubleStream for S where S: Stream<Item = i32> {}
```

4.9. Distributing your operator

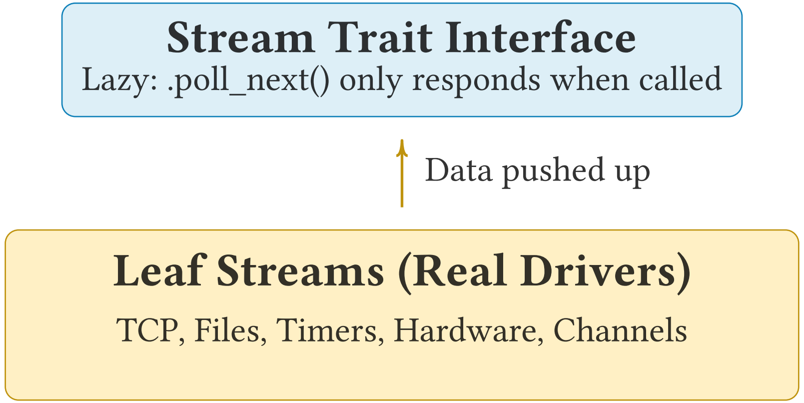
Define a constructor and turn it into a method of an **extension trait**:

```rust
trait DoubleStream: Stream {
    fn double(self) -> Double<Self>
    where Self: Sized + Stream<Item = i32>,
    { Double::new(self) }
}
  // A blanket implementation should be provided by you!
  impl<S> DoubleStream for S where S: Stream<Item = i32> {}
```

Now, users **don’t need to know how** `Double` is implemented, just

1. import your extension trait: `DoubleStream`
2. call `.double()` on any compatible stream

4.10. The ‘*real*’ stream drivers



`Stream` trait just provides a **uniform way to query** - it doesn’t create or drive data flow.

4.10. The *'real'* stream drivers

The `Stream` trait is NOT the stream itself - it's just a lazy frontend to query data.

What `Stream` trait does:

- Provides uniform `.poll_next()` interface
- Lazy: only responds when asked
- Doesn't drive or produce data itself
- Just queries whatever backend exists

What actually drives streams:

- TCP connections receiving packets
- File I/O completing reads
- Timers firing
- Hardware signals
- Channel senders pushing data

5.1. Complexity $1 \to N$ operators

Challenges for `Stream` operators are combined from:

Inherent `Future` challenges:

- Clean up orphaned wakers
- Cleanup when tasks abort
- Task coordination complexity

5.1. Complexity $1 \to N$ operators

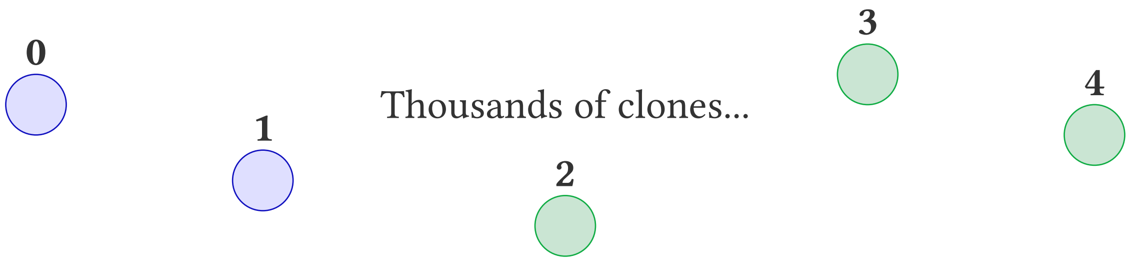
Challenges for `Stream` operators are combined from:

Inherent `Future` challenges:

- Clean up orphaned wakers
- Cleanup when tasks abort
- Task coordination complexity

Inherent `Iterator` challenges:

- Ordering guarantees across consumers
- Backpressure with slow consumers
- Sharing mutable state safely
- Avoiding duplicate items



All in different states

5.2. Sharing latency between tasks

Latency may need to processed by different async tasks:

```rust
let tcp_stream = TcpStream::connect("127.0.0.1:8080").await?;
let latency = tcp_stream.latency(); // Stream<Item = Duration>

spawn(async move { display_ui(latency).await; });
spawn(async move { engage_breaks(latency).await; }); // Error!
```

Error

`latency` is moved into the first task, so the second task can't access it.

Warning

We need a way to clone the latency stream!

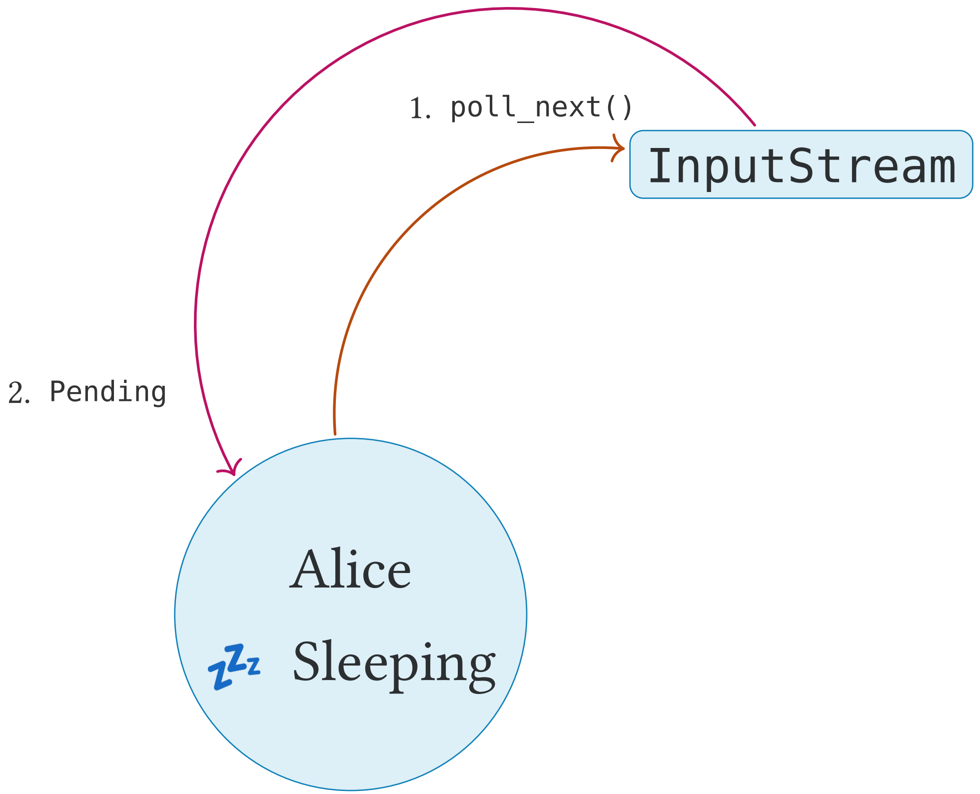
5.2. Sharing latency between tasks

Solution: Create a ***stream operator*** `fork()` makes the input stream `Clone`.

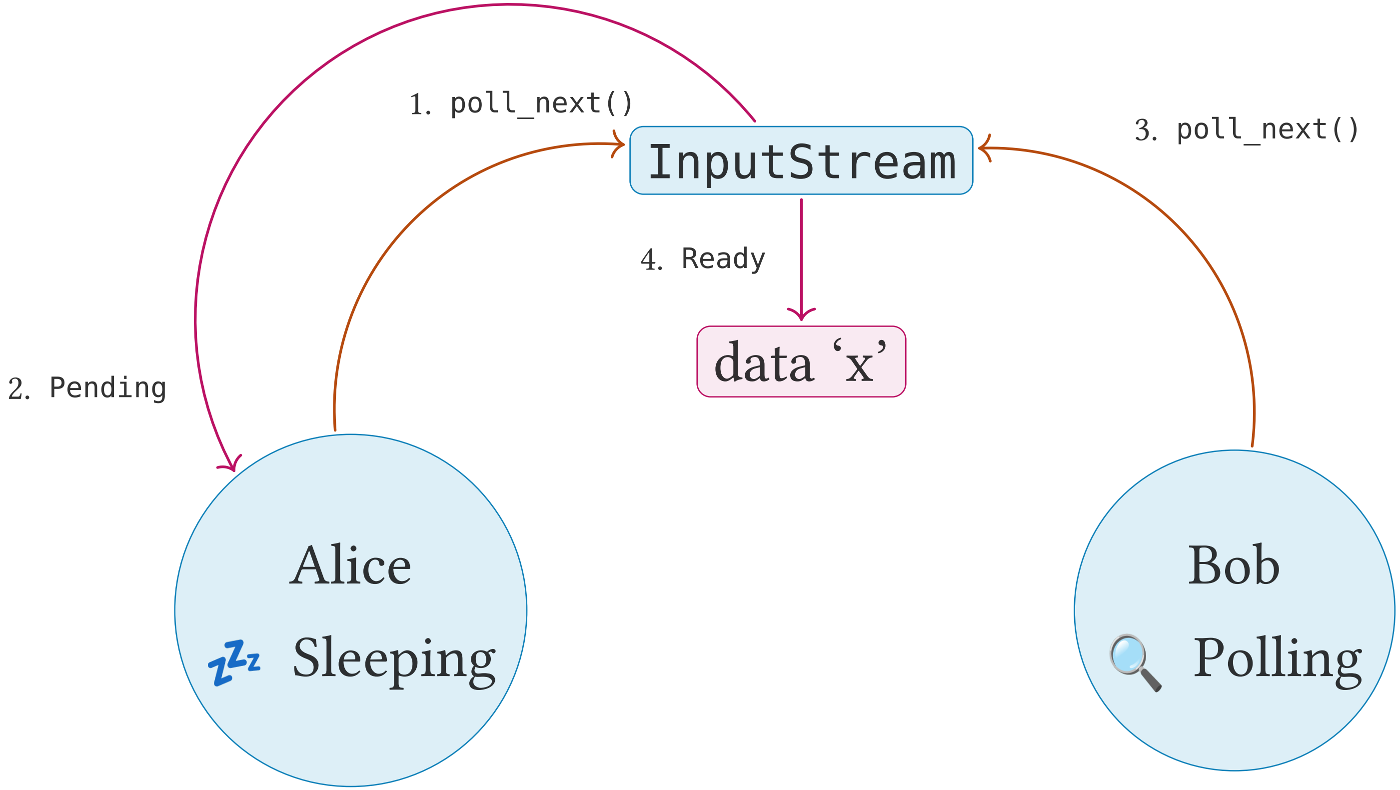
```rust
let ui_latency = tcp_stream.latency().fork();

let breaks_latency_clone = ui_latency.clone();
// Warning: `Clone` needs to be implemented!

  spawn(async move { display_ui(ui_latency).await; });
  spawn(async move { engage_breaks(breaks_latency_clone).await; });
```

Requirement: `Stream<Item: Clone>`, so we can clone the items (`Duration` is `Clone`)

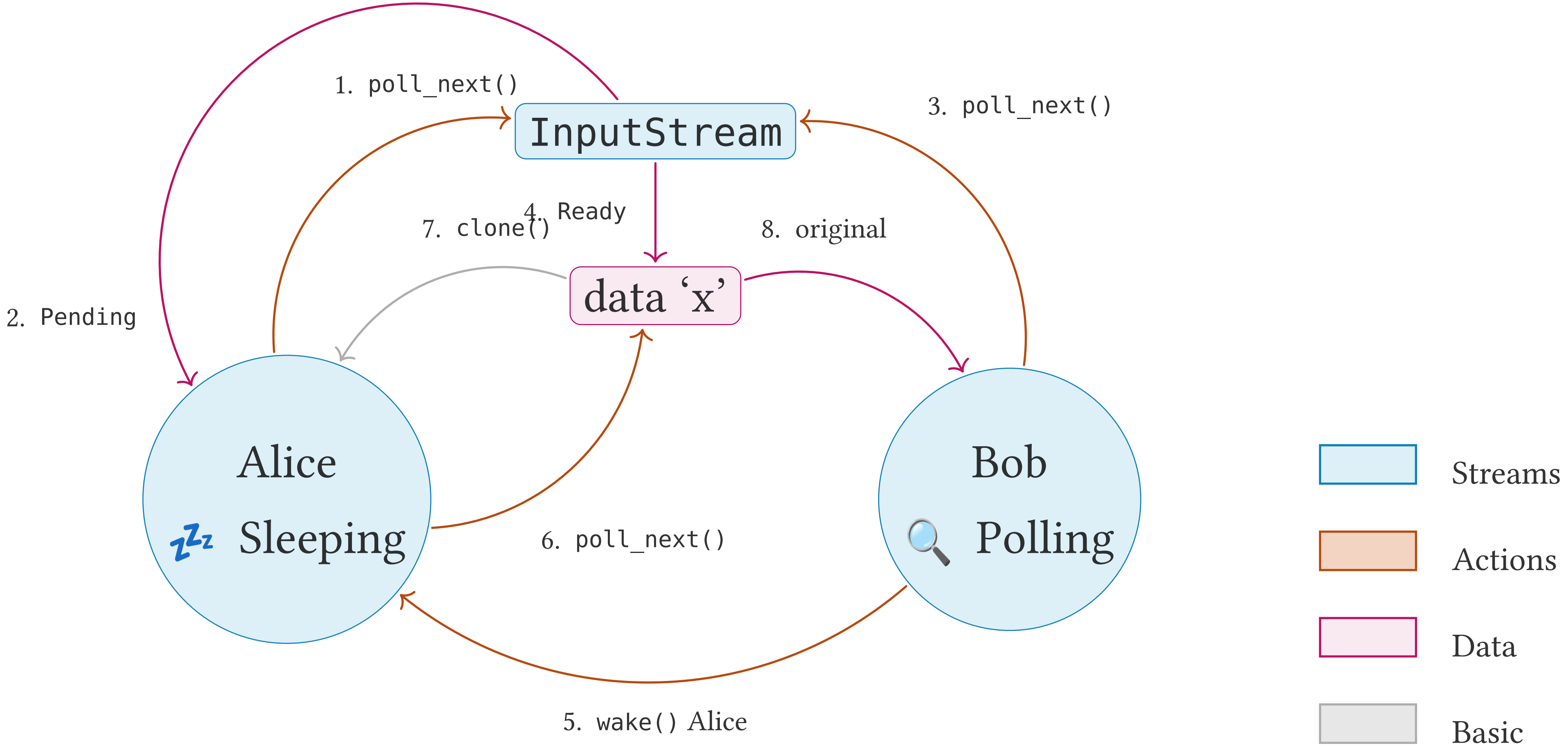
5.3. Handling sleeping and waking



5.3. Handling sleeping and waking



5.3. Handling sleeping and waking

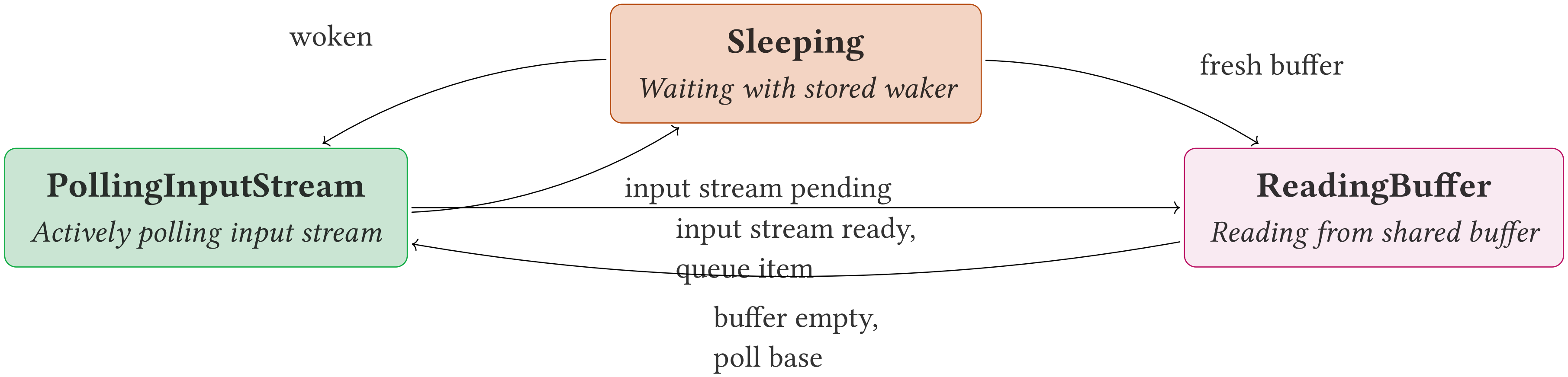


5.4. Simplified state machine of `clone-stream`

Enforcing simplicity, **correctness and performance**:

Solution

Each clone maintains its own state

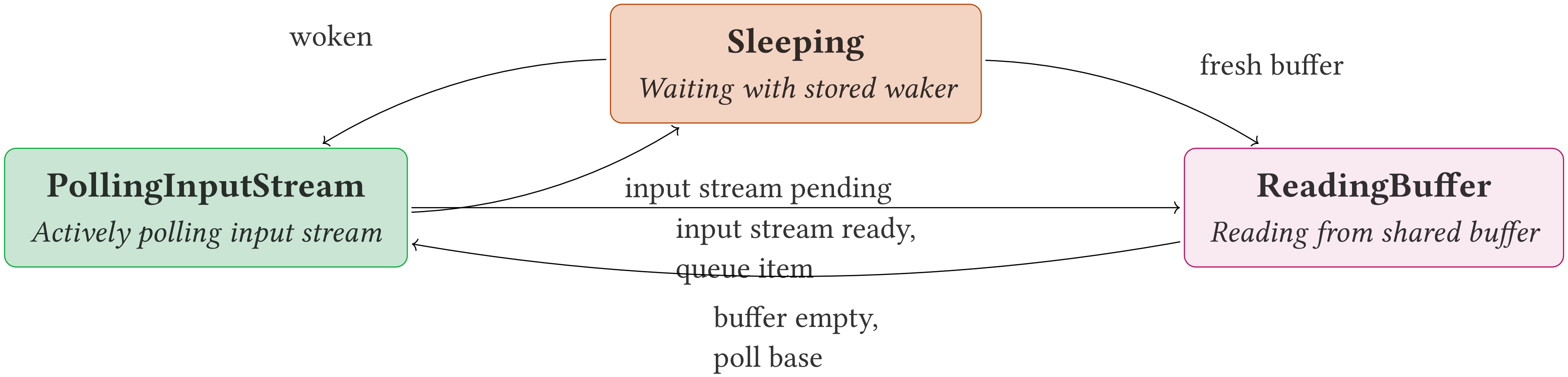


5.4. Simplified state machine of `clone-stream`

Enforcing simplicity, **correctness and performance**:

Solution

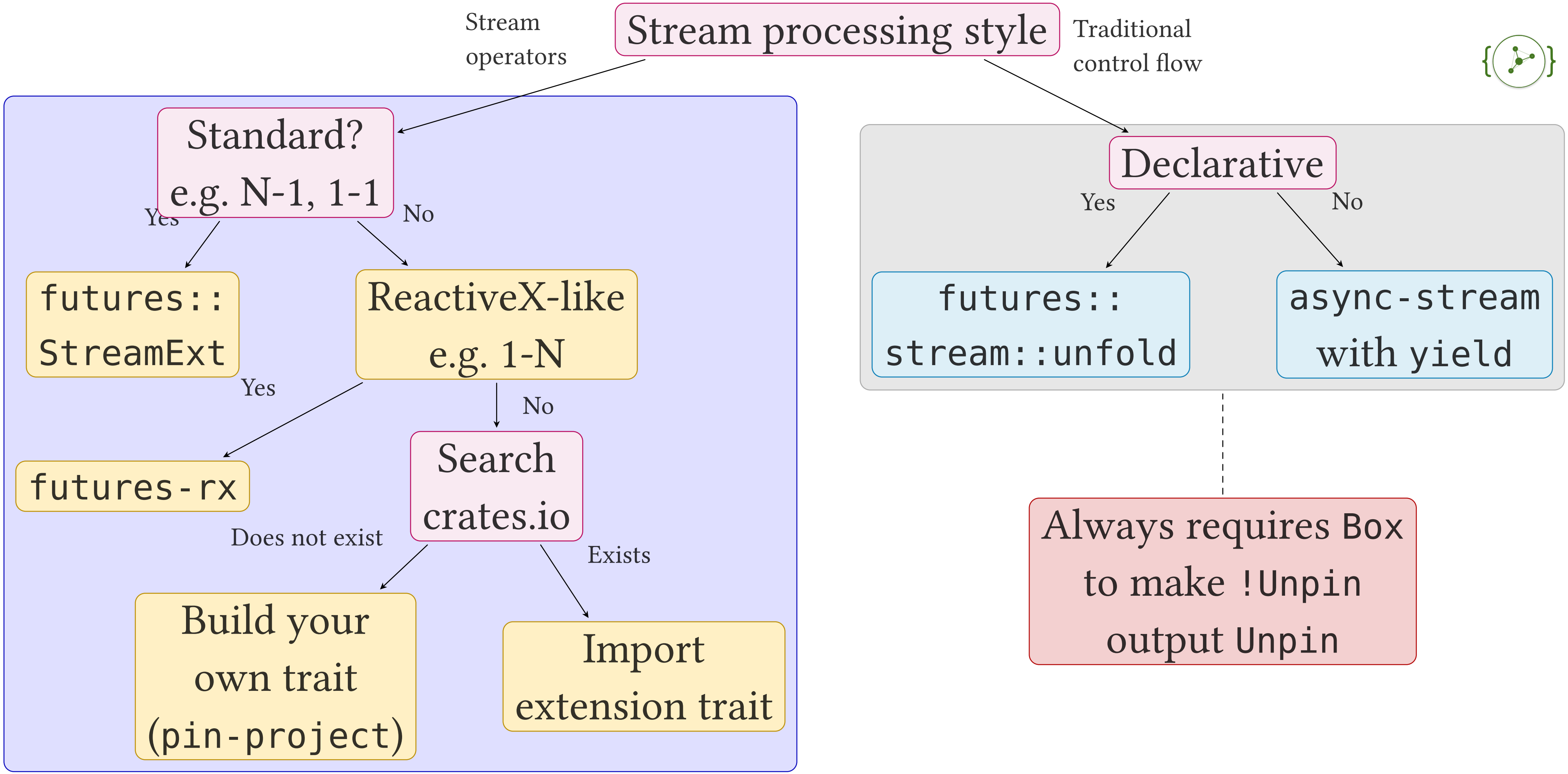
Each clone maintains its own state



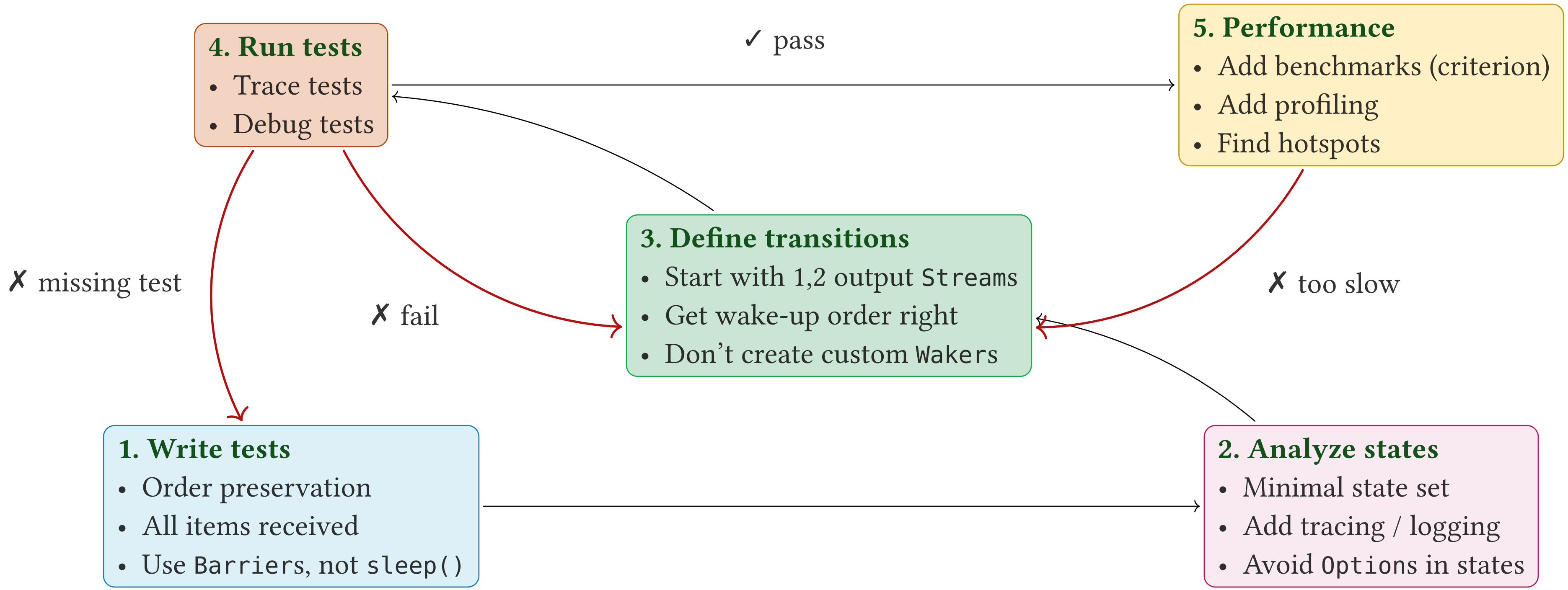
Speed

8 - 12 micro seconds per item per clone. (Using `pin-project` did not speed up.)

6.1. Quickstart



6.2. Advanced operator construction



6.3. Questions

Thank you for your attention!

- Contact me: willemvanhulle@protonmail.com
- These slides: github.com/wvhulle/streams-eurorust-2025

Learn more?

Join my 7-week course ***"Creating Safe Systems in Rust"***

- Location: Ghent (Belgium)
- Date: starting 4th of November 2025.

Register at pretix.eu/devlab/rust-course/