



Transforming Streams

Advanced stream processing in Rust

Willem Vanhulle

EuroRust 2025

1980-01-01

1. Introduction	1
1.1. Me	2
1.2. Kinds of streams	3
1.3. Streams in Rust are not new	4
1.4. Why does Rust need special treatment?	5
1.5. Process TCP connections and collect long messages	6
1.6. Stream operators: declarative & composable	7
2. Rust's Stream trait	8
3. Using Streams	14
4. Example 1: $1 \rightarrow 1$ Operator	20
5. Example 2: $1 \rightarrow N$ Operator	44
6. Conclusion	50



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



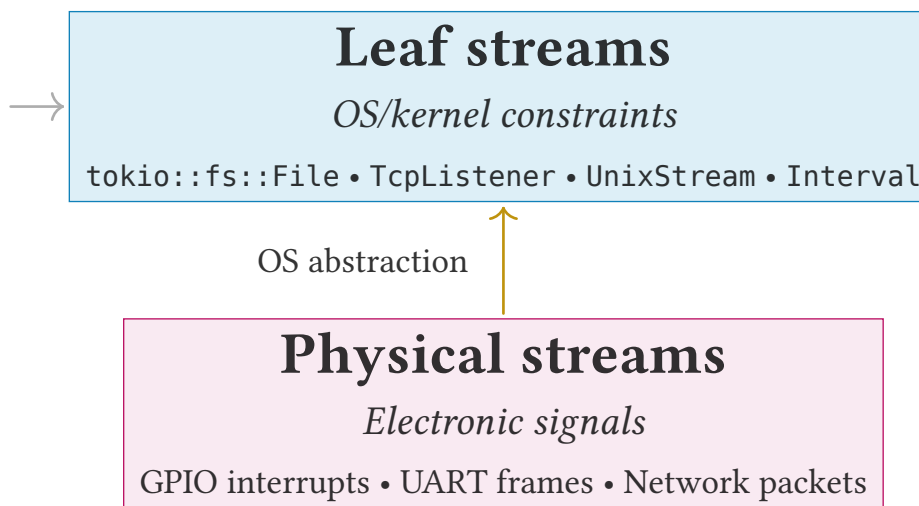
Physical streams

Electronic signals

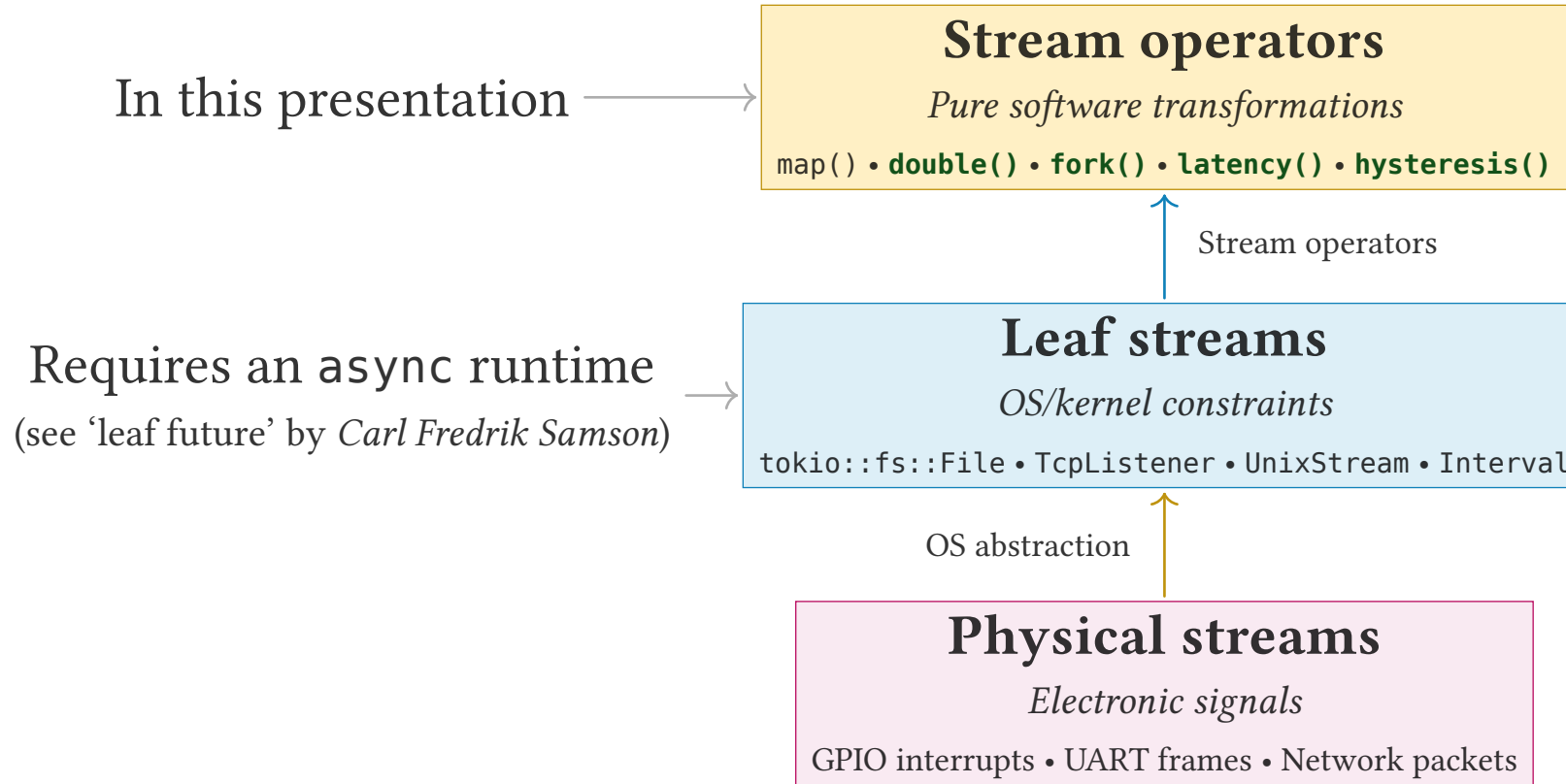
GPIO interrupts • UART frames • Network packets



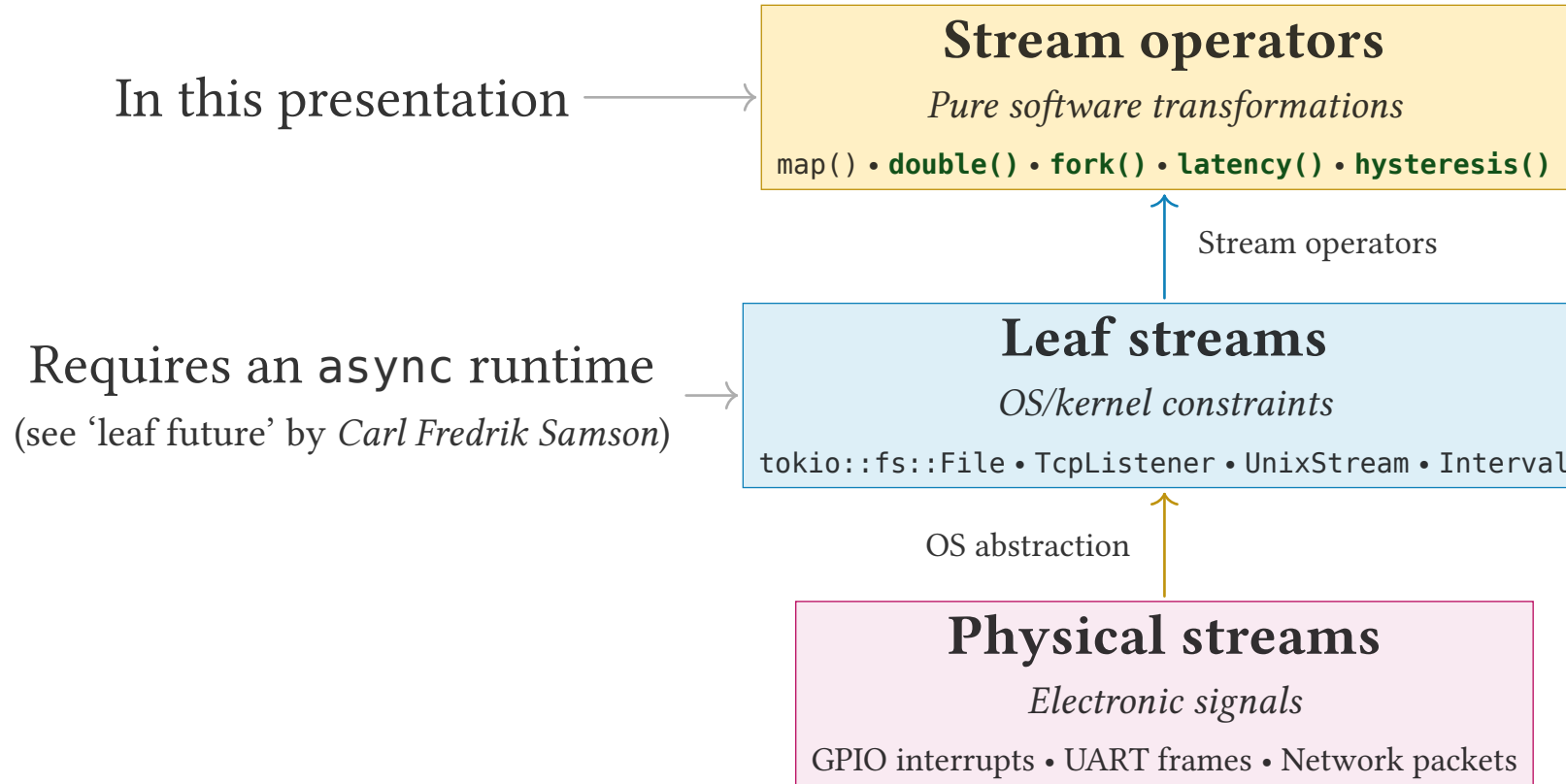
Requires an async runtime
(see 'leaf future' by *Carl Fredrik Samson*)



1.2. Kinds of streams



1.2. Kinds of streams



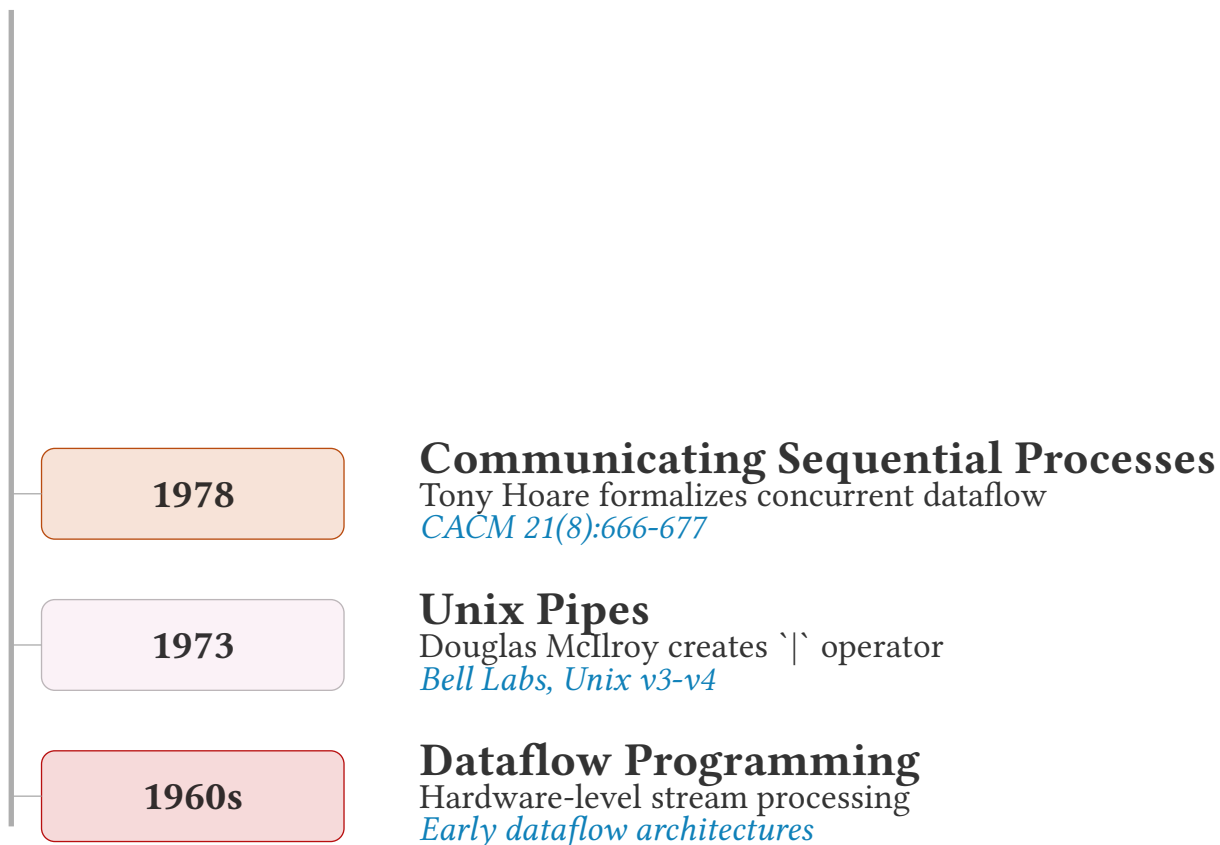
Operators 

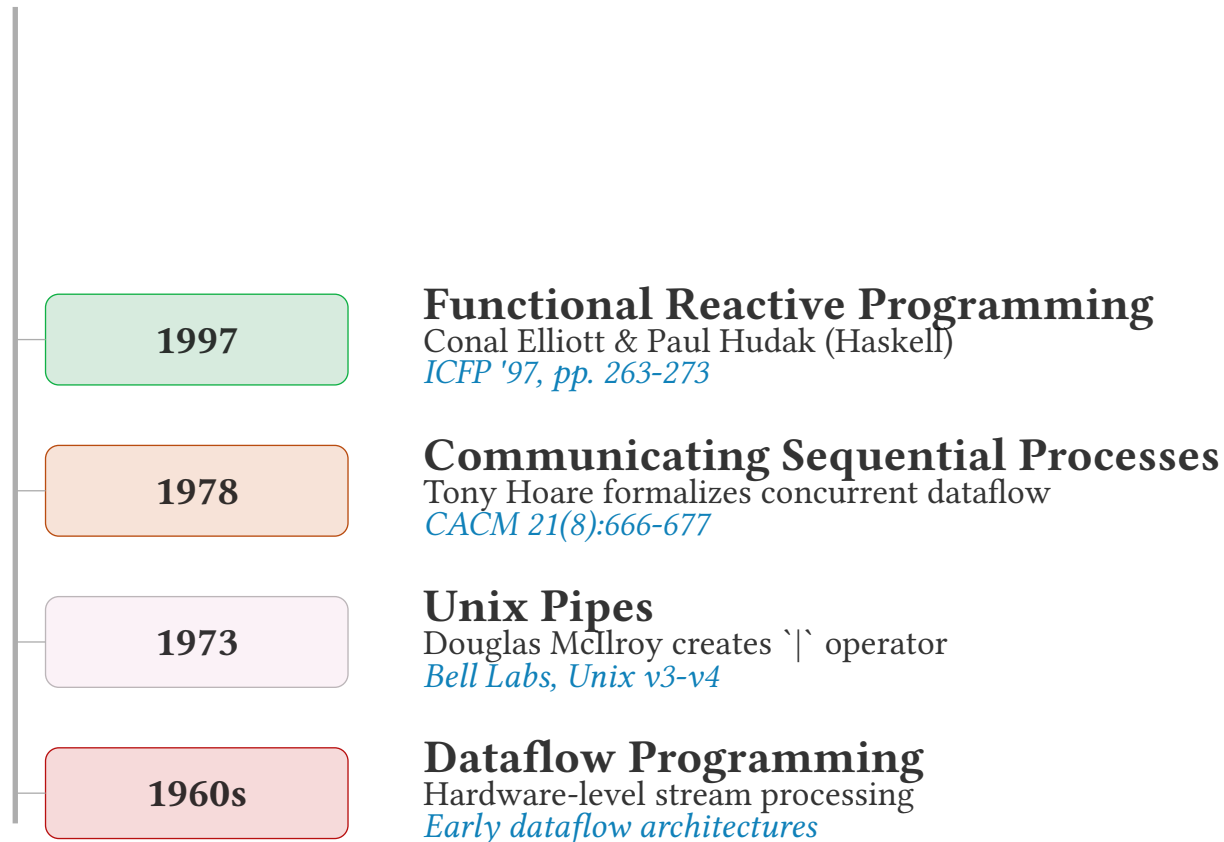
Streams

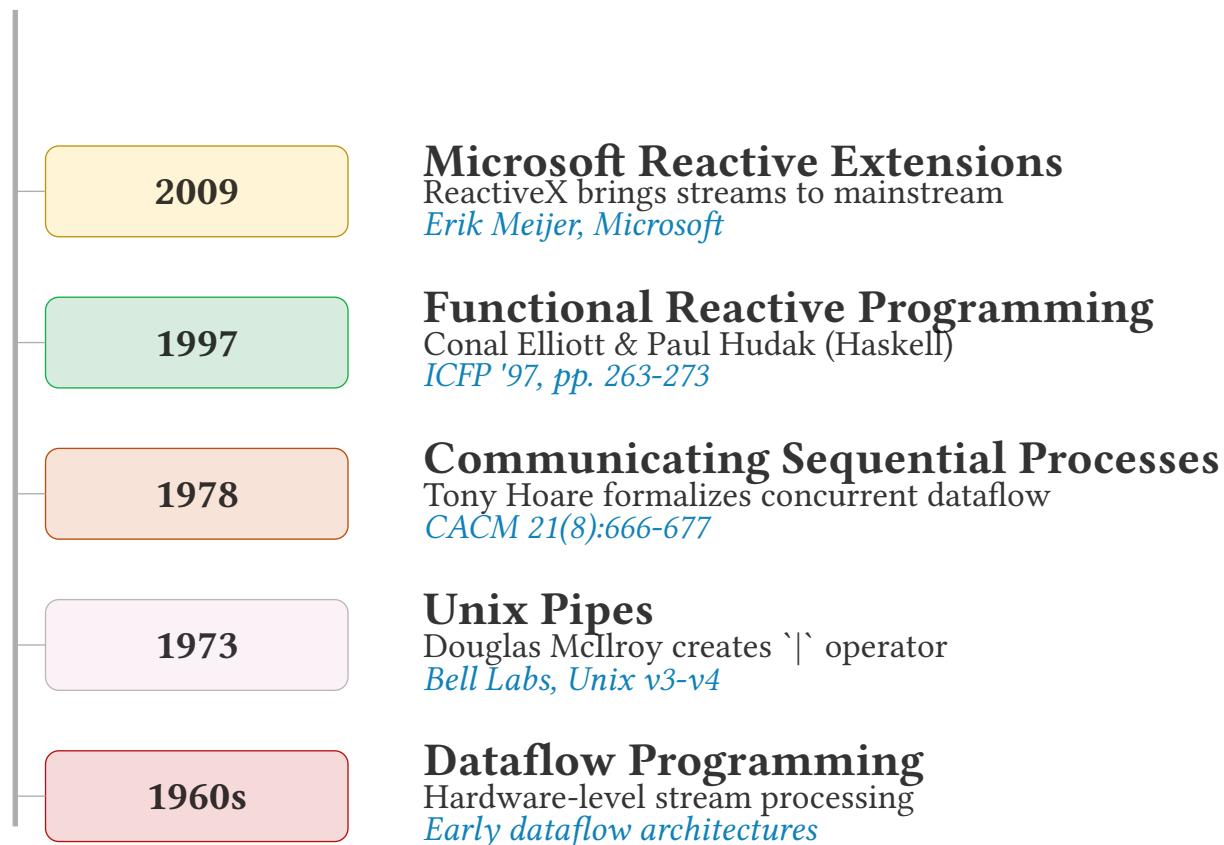
Data

Hardware signals are abstracted by the OS

Software operators transform the streams



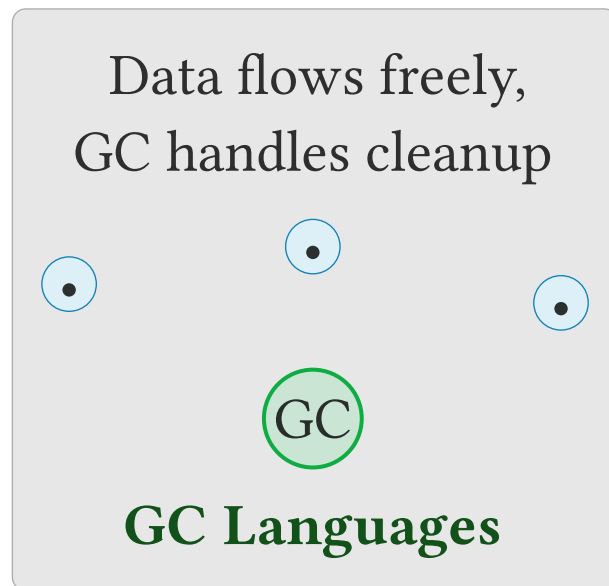




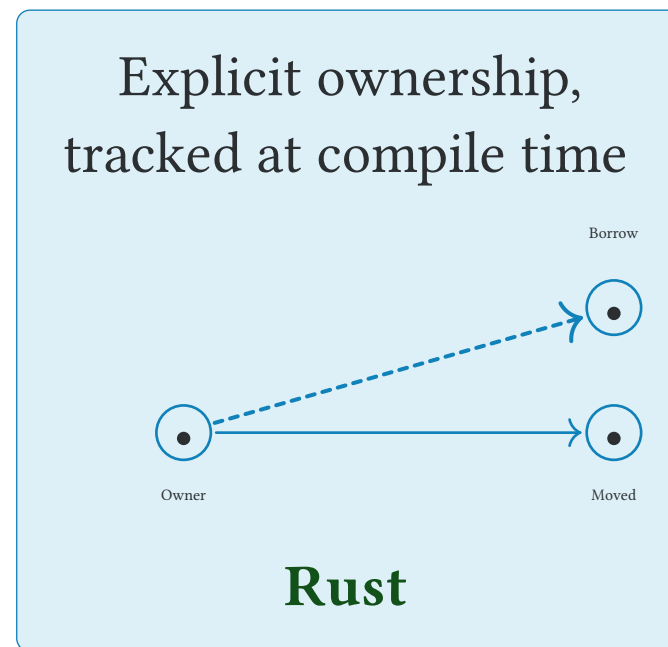


1.4. Why does Rust need special treatment?

Stream operators must wrap and own their input by value



vs



1.5. Process TCP connections and collect long messages



```
1  let mut results = Vec::new(); let mut count = 0;
2
3  while let Some(connection) = tcp_stream.next().await {
4      match connection {
5          Ok(stream) if should_process(&stream) => {
6              match process_stream(stream).await {
7                  Ok(msg) if msg.len() > 10 => {
8                      results.push(msg);
9                      count += 1;
10                     if count >= 5 { break; }
11                 }
12                 Ok(_) => continue,
13                 Err(_) => continue,
14             }
15         }
16         Ok(_) => continue,
17         Err(_) => continue,
18     }
19 }
```



Problems:

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



1.6. Stream operators: declarative & composable

Same logic with stream operators:

```
1 let results: Vec<String> = tcp_stream
2   .filter_map(|conn| ready(conn.ok()))
3   .filter(|stream| ready(should_process(stream)))
4   .then(|stream| process_stream(stream))
5   .filter_map(|result| ready(result.ok()))
6   .filter(|msg| ready(msg.len() > 10))
7   .take(5)
8   .collect()
9   .await;
```

 Rust

Benefits:

- Each operation is isolated
- Testable
- Reusable

“Programs must be written **for people to read**”

1. Introduction	1
2. Rust's Stream trait	8
2.1. Moving from Iterator to Stream	9
2.2. The Stream trait: async iterator	10
2.3. Possible inconsistency	11
2.4. The meaning of Ready(None)	12
2.5. 'Fusing' Streams and Futures	13
3. Using Streams	14
4. Example 1: $1 \rightarrow 1$ Operator	20
5. Example 2: $1 \rightarrow N$ Operator	44
6. Conclusion	50

2.1. Moving from Iterator to Stream

✓ Always returns immediately

⚠ May be Pending

2. Rust's Stream trait

✓ 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)

2.1. Moving from Iterator to Stream

✓ Always returns immediately

⚠ May be Pending

✓ Hides polling complexity



next() → Some(3)

poll_next() → Ready(Some(2))

next() → Some(1)

poll_next() → Pending

next() → None

poll_next() → Ready(Some(1))

next() → Some(2)

poll_next() → Pending

Iterator (sync)

Stream (low-level)

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)

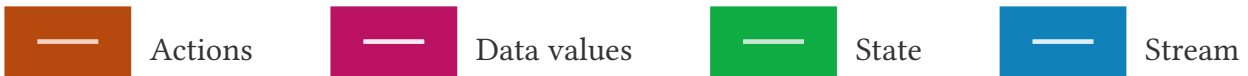
next().await → Some(3)

next().await → Some(1)

next().await → None

next().await → Some(2)

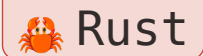
Stream (high-level)



2.2. The Stream trait: async iterator

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

```
1 trait Stream {  
2     type Item;  
3  
4     fn poll_next(self: Pin<&mut Self>, cx: &mut Context<'_>)  
5         -> Poll<Option<Self::Item>>;  
6 }
```



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

```
1 trait Stream {  
2     type Item;  
3  
4     fn poll_next(self: Pin<&mut Self>, cx: &mut Context)  
5         -> Poll<Option<Self::Item>>  
6 }
```



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.



Regular Stream

“No items **right now**”

(Stream might yield more later)

Fused Stream

“No items **ever again**”

(Stream is permanently done)

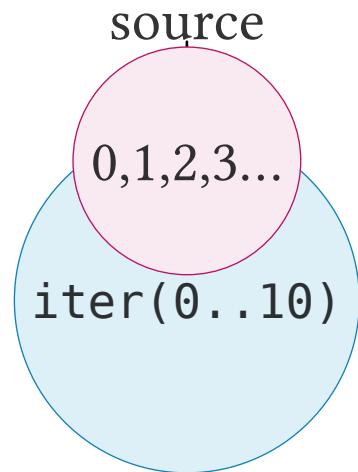


	Future	Stream	Meaning
Regular			May continue
Fused	FusedFuture	FusedStream	is_terminated() method
Fused			Done permanently
Fused value	Pending	Ready (None)	Final value

1.	Introduction	1
2.	Rust's <code>Stream</code> trait	8
3.	Using Streams	14
3.1.	Pipelines with <code>futures::StreamExt</code>	15
3.2.	The handy <code>std::future::ready</code> function	16
3.3.	Flatten a finite collection of <code>Streams</code>	17
3.4.	Flattening an infinite stream	18
3.5.	More <code>Stream</code> features to explore	19
4.	Example 1: $1 \rightarrow 1$ Operator	20
5.	Example 2: $1 \rightarrow N$ Operator	44
6.	Conclusion	50

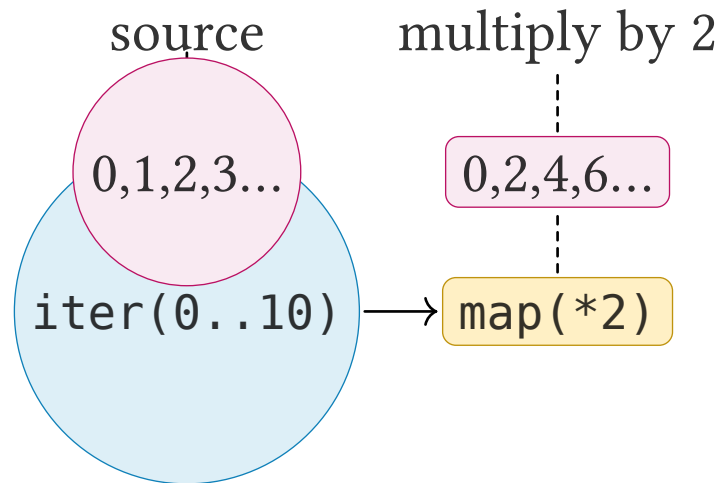
3.1. Pipelines with futures::StreamExt

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



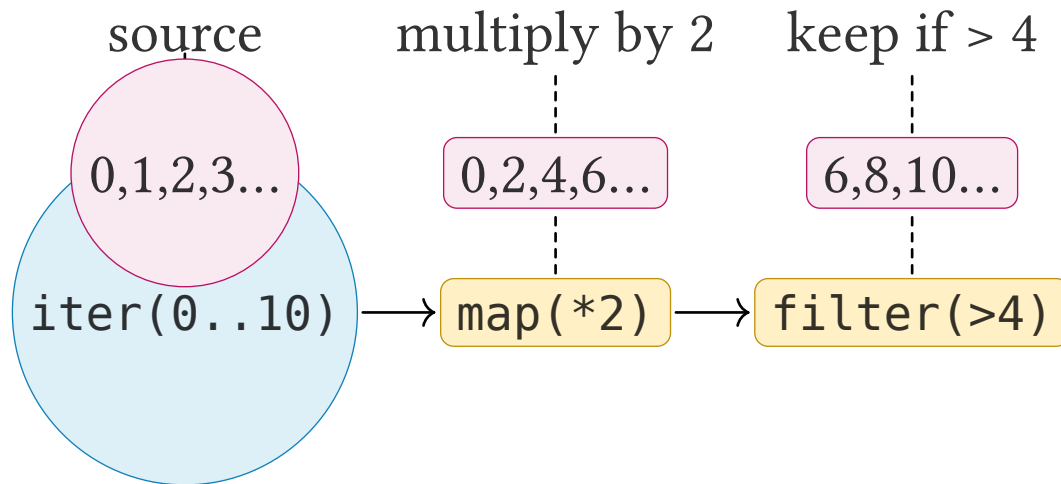
3.1. Pipelines with futures::StreamExt

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



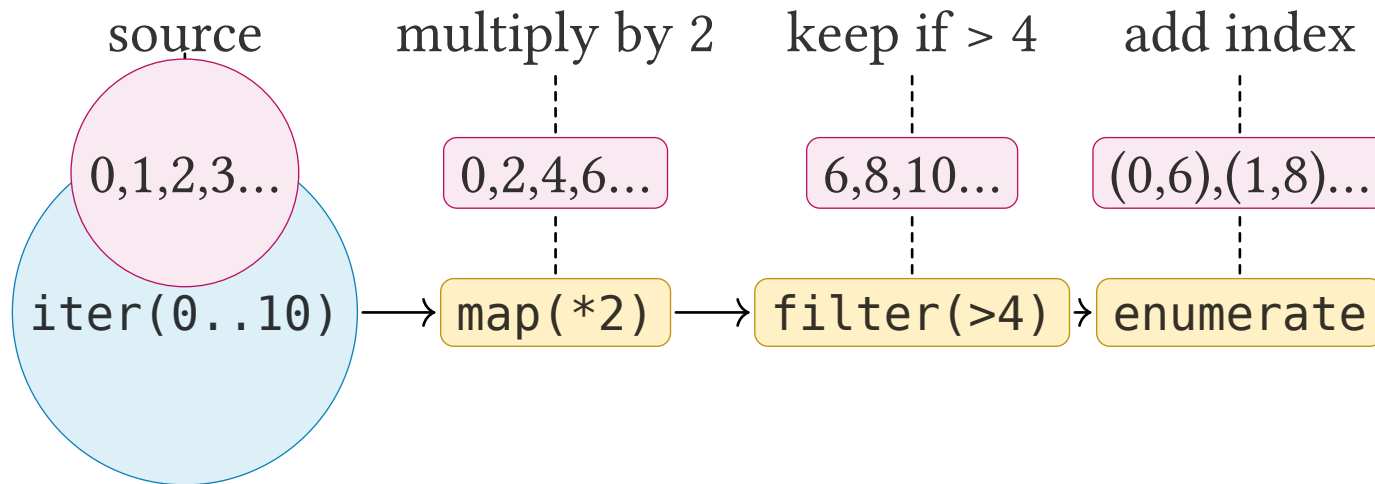
3.1. Pipelines with futures::StreamExt

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



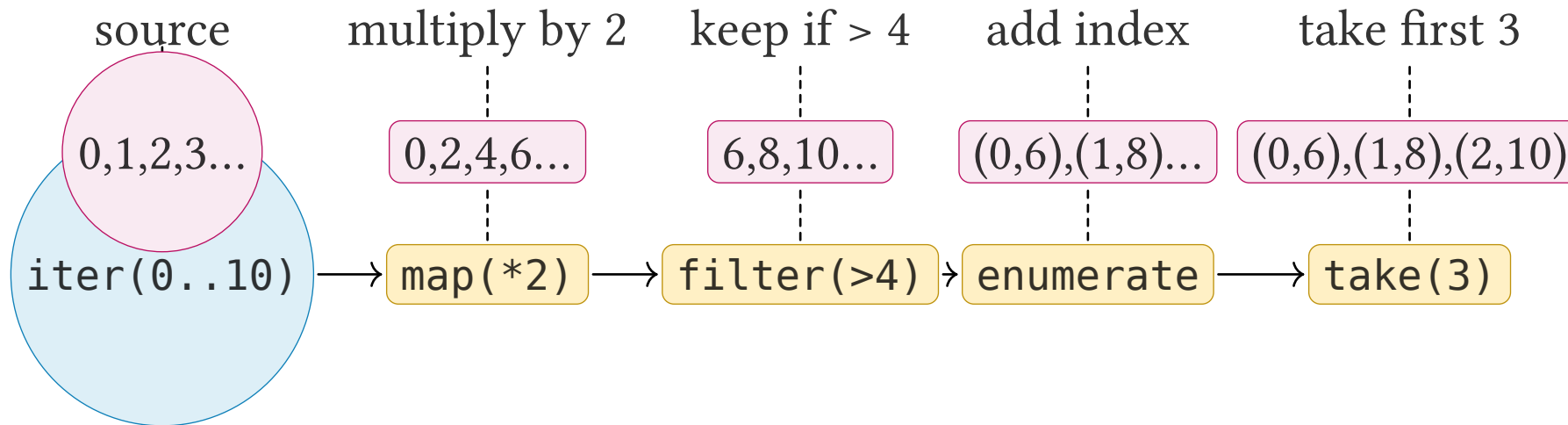
3.1. Pipelines with futures::StreamExt

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



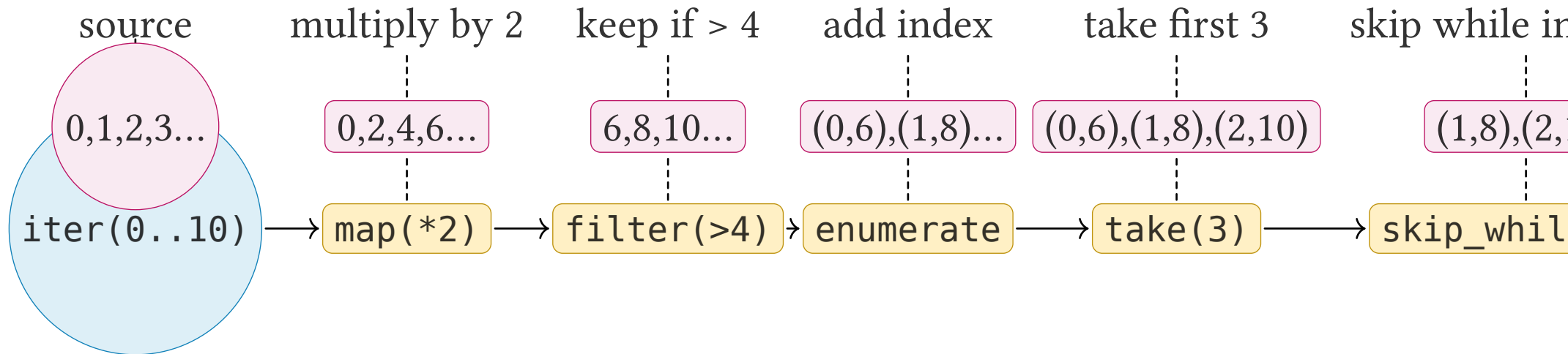
3.1. Pipelines with futures::StreamExt

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

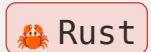


3.1. Pipelines with futures::StreamExt

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



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



Rust

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!)

```
1 stream.filter(|&x| async move {
2     x % 2 == 0
3 })
```



Rust

Option 2: Async closure (not Unpin!)

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



Rust

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

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



Rust

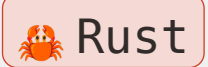
- `ready(value)` creates a `Future` that immediately resolves to `value`.
- `ready(value)` is `Unpin`

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>`



```
1 let streams = vec![  
2     stream::iter(1..=3),  
3     stream::iter(4..=6),  
4     stream::iter(7..=9),  
5 ];  
6  
7 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!



```
1 let infinite_streams = stream::unfold(0, |id| async move {  
2     Some((stream::iter(id..), id + 1))  
3 });  
4 let flat = infinite_streams.flatten();
```



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

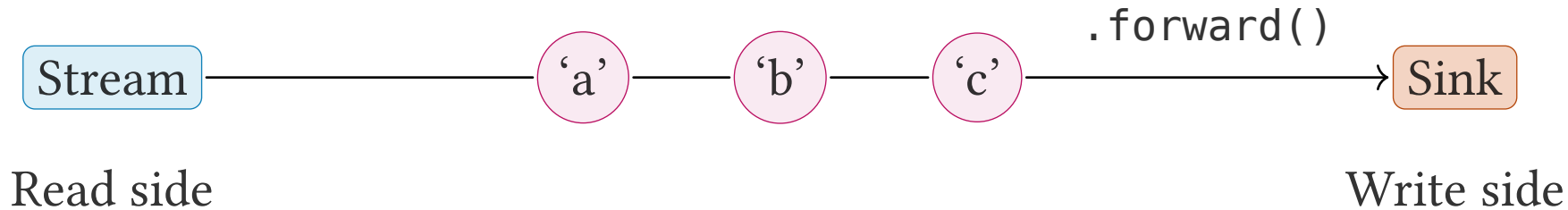
```
1 let requests = stream::unfold(0, |id| async move {  
2     Some((fetch_stream(format!("/api/data/{}", id)), id + 1))  
3 });  
4 let flat = requests.flatten_unordered(Some(10));
```



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

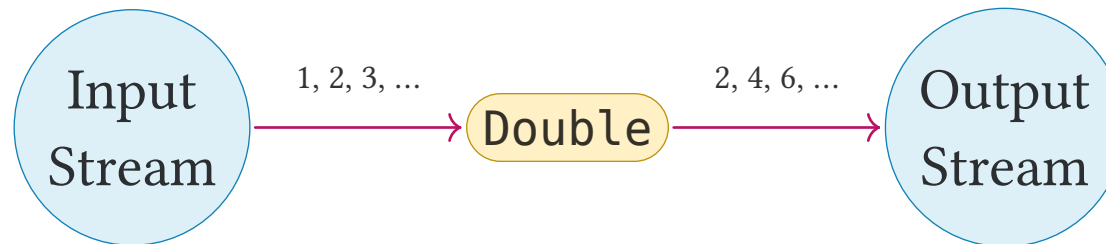


1. Introduction	1
2. Rust's Stream trait	8
3. Using Streams	14
4. Example 1: $1 \rightarrow 1$ Operator	20
4.1. Doubling stream operator	22
4.2. Naive implementation of <code>poll_next</code>	25
4.3. Why does <code>Pin::get_mut()</code> require <code>Unpin</code> ? ...	28
4.4. <code>Unpin</code> types can be safely unpinned	29
4.5. <code>!Unpin</code> types cannot be unpinned	31
4.6. Following compiler hints	33
4.7. Turning <code>!Unpin</code> into <code>Unpin</code> with boxing	35
4.8. Review of approaches to <code>!Unpin</code> fields	39
4.9. Distributing your operator	41
4.10. The ‘ <i>real</i> ’ stream drivers	42

5.	Example 2: $1 \rightarrow N$ Operator	44
6.	Conclusion	50



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

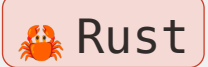


Input stream **needs to yield integers**.

4.1. Doubling stream operator

Step 1: Define a struct that wraps the input stream

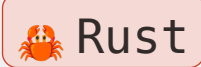
```
1 struct Double<InSt> {  
2     in_stream: InSt,  
3 }
```



- 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



```
1  impl<InSt> Stream for Double<InSt>
2  where
3      InSt: Stream<Item = i32>
4  {
5      type Item = i32;
6
7      fn poll_next(self: Pin<&mut Self>, cx: &mut Context<'_>)
8          -> Poll<Option<Self::Item>> {
9          // ... implementation goes here
10     }
11 }
```

4.2. Naive implementation of poll_next

4. Example 1: $1 \rightarrow 1$ Operator

Focus on the implementation of the poll_next method



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

```
1 fn poll_next(self: Pin<&mut Self>, cx: &mut Context<'_>)  
2     -> Poll<Option<Self::Item>> {  
3     // Cannot access self.in_stream!  
4     Pin::new(&mut self.in_stream) // Not possible!  
5     .poll_next(cx)  
6     .map(|x| x * 2)  
7 }
```



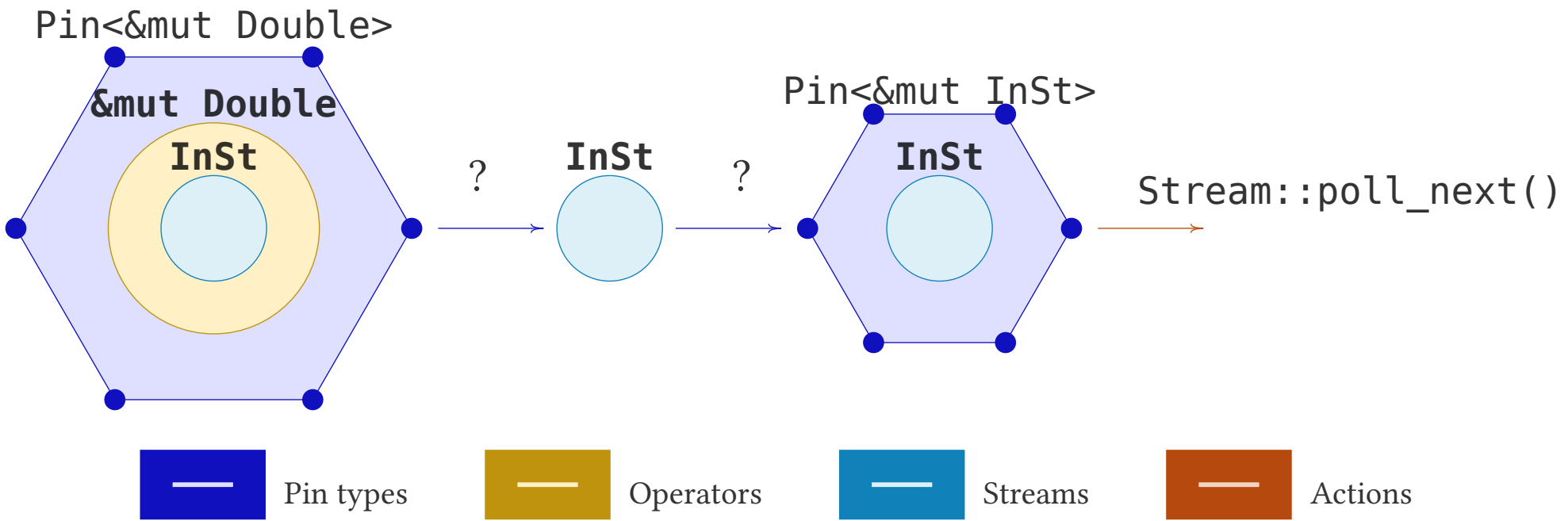
Pin<&mut Self> **blocks access to self.in_stream** (when Self: !Unpin)!

4.2. Naive implementation of poll_next

Warning

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?

```

1  impl<InSt> Stream for Double<InSt> where InSt: Stream<Item = i32> {
2
3      type Item = InSt::Item;
4
5      fn poll_next(self: Pin<&mut Self>, cx: &mut Context<'_>)
6          -> Poll<Option<Self::Item>> {
7          let this = self.get_mut(); // Error!
8          let pinned_in = Pin::new(&mut this.in_stream);
9          pinned_in.poll_next(cx).map(|p| p.map(|x| x * 2))
10     }
11     }

```



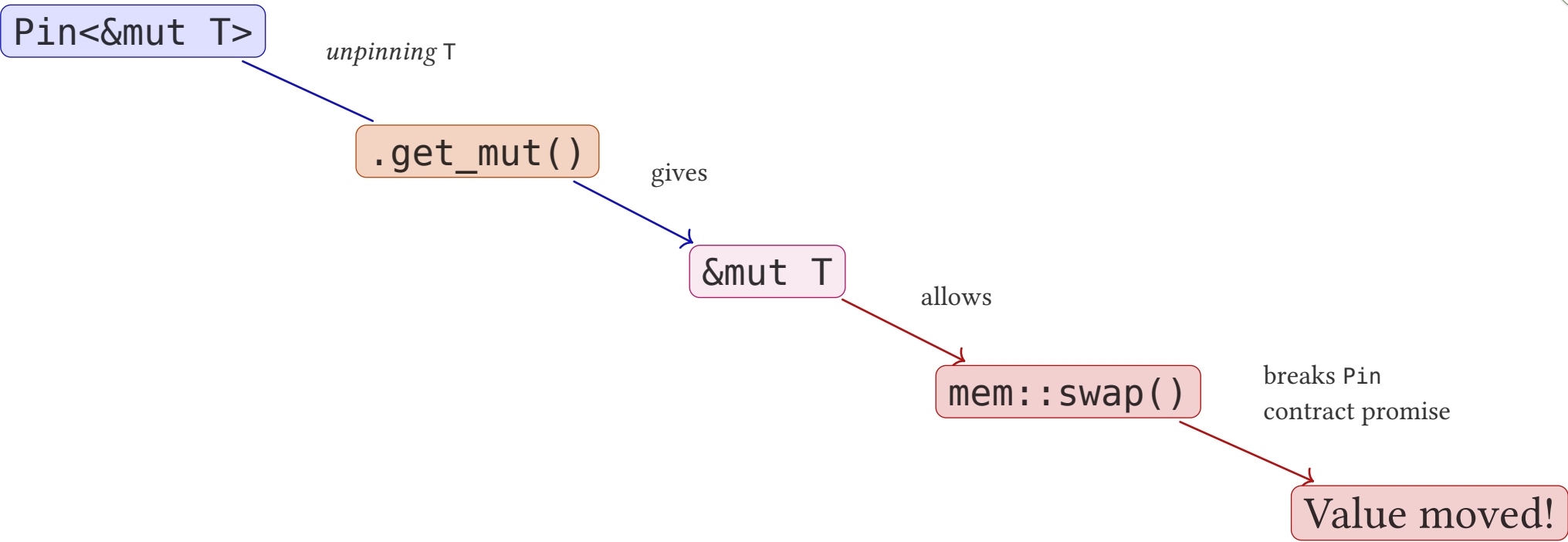
Problem: `Pin::get_mut()` requires `Double<InSt>: Unpin`

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

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

4. Example 1: $1 \rightarrow 1$ Operator

`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

4. Example 1: 1 → 1 Operator



Unpin Bird

Safe to move

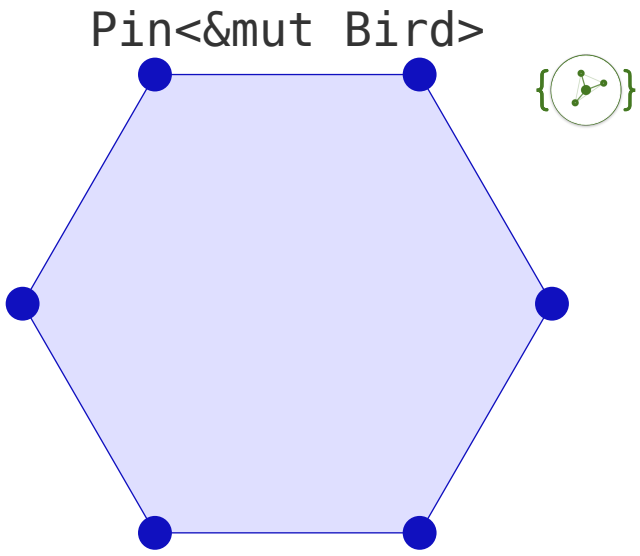
4.4. Unpin types can be safely unpinned

4. Example 1: 1 → 1 Operator



Unpin Bird

Safe to move

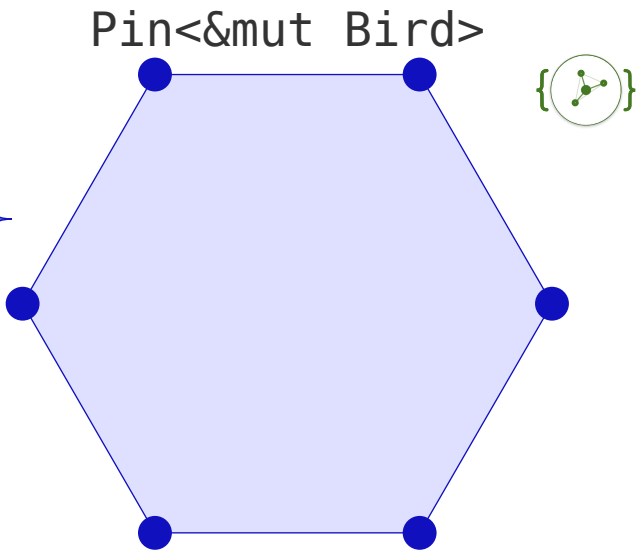


4.4. Unpin types can be safely unpinned

4. Example 1: $1 \rightarrow 1$ Operator


Unpin Bird
Safe to move

`Pin::new()`

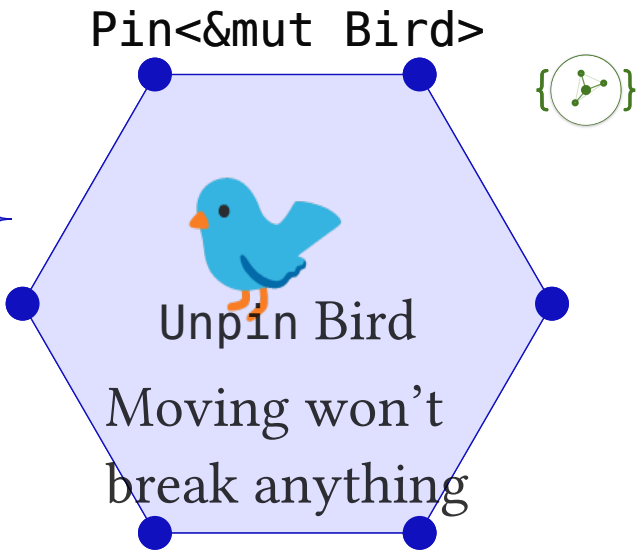


4.4. Unpin types can be safely unpinned

4. Example 1: $1 \rightarrow 1$ Operator

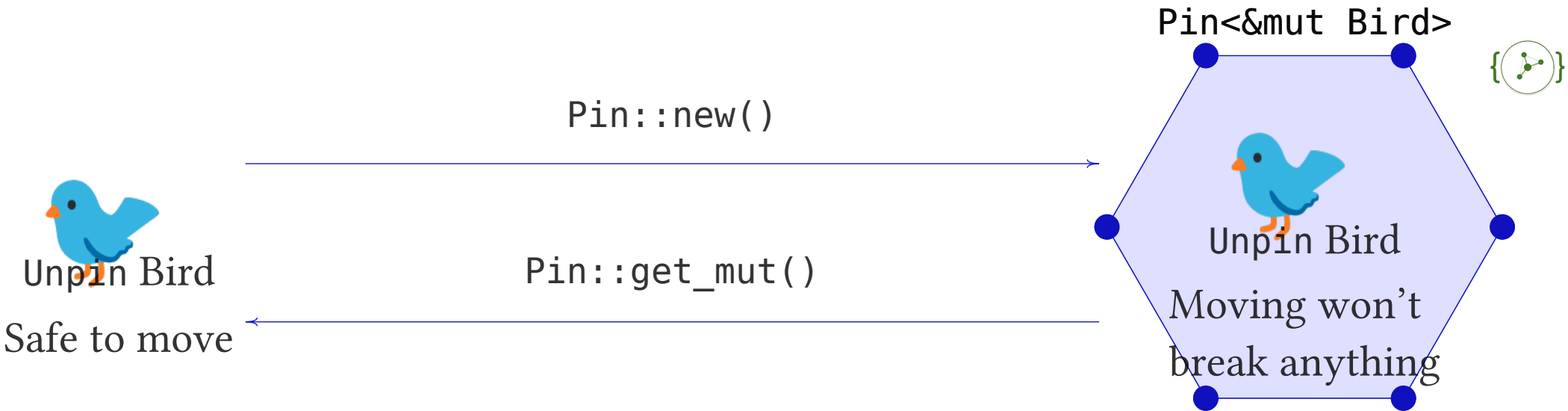

Unpin Bird
Safe to move

`Pin::new()`



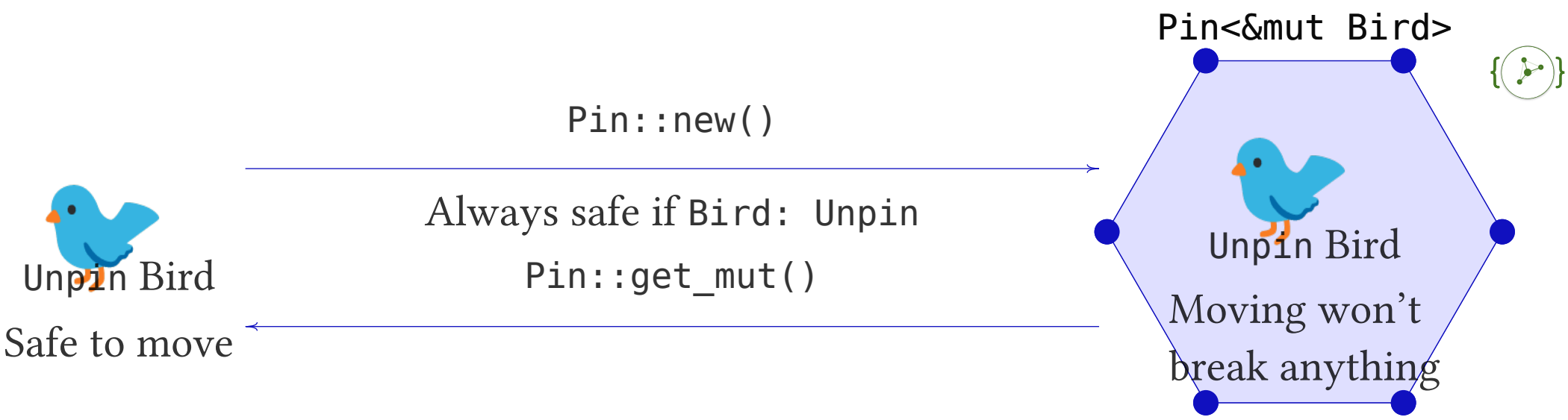
4.4. Unpin types can be safely unpinned

4. Example 1: $1 \rightarrow 1$ Operator



4.4. Unpin types can be safely unpinned

4. Example 1: $1 \rightarrow 1$ Operator



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

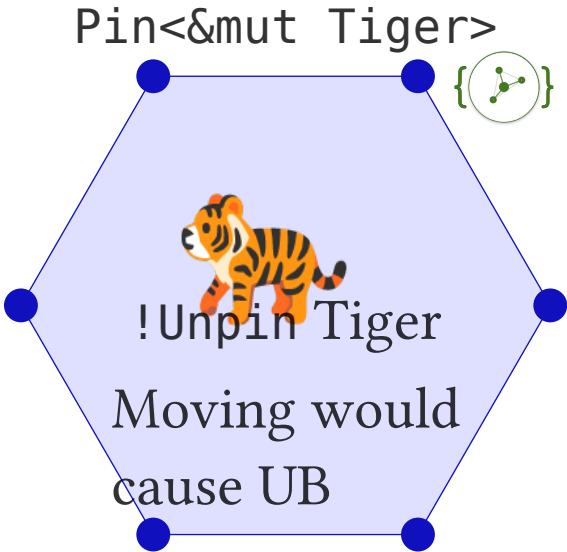


!Unpin Tiger

Dangerous to move

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

Would break
pin promise!



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`:



```
1  impl<InSt> Stream for Double<InSt> where InSt: Stream<Item = i32> + Unpin {
2      type Item = InSt::Item;
3
4      fn poll_next(self: Pin<&mut Self>, cx: &mut Context<'_>) -> Poll<Option<Self::Item>> {
5          // `this` = a conventional name for `get_mut` output
6          let mut this = self.get_mut();
7          let pinned_in = Pin::new(&mut this.in_stream);
8          pinned_in
9              .poll_next(cx)
10             .map(|p| p.map(|x| x * 2))
11     }
12 }
```



4.6. Following compiler hints

The compiler error suggests adding `InSt: Unpin`:

```
1  impl<InSt> Stream for Double<InSt> where InSt: Stream<Item = i32> + Unpin {  
2      type Item = InSt::Item;  
3  
4      fn poll_next(self: Pin<&mut Self>, cx: &mut Context<'_>) -> Poll<Option<Self::Item>> {  
5          // `this` = a conventional name for `get_mut` output  
6          let mut this = self.get_mut();  
7          let pinned_in = Pin::new(&mut this.in_stream);  
8          pinned_in  
9              .poll_next(cx)  
10             .map(|p| p.map(|x| x * 2))  
11     }  
12 }
```



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()`

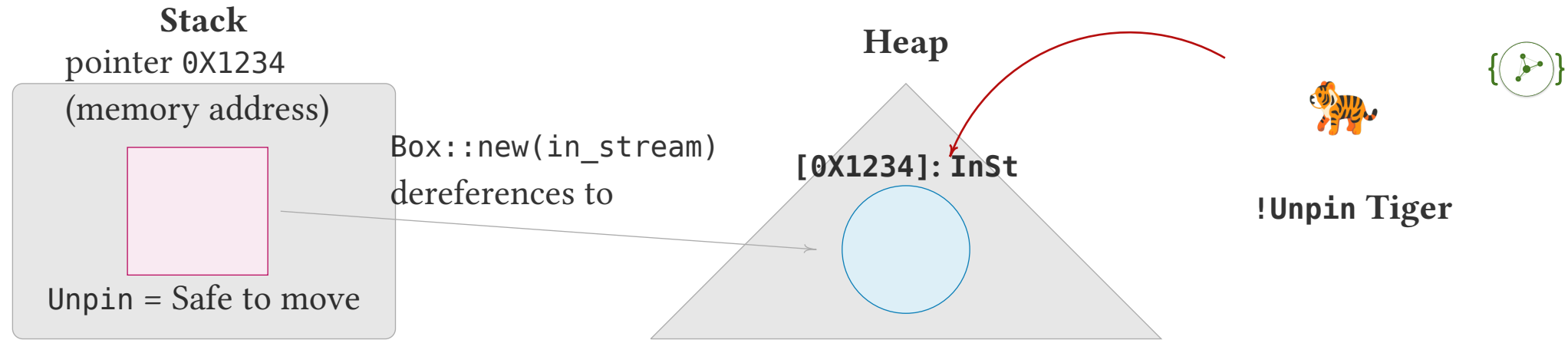
Instead of forcing customers of our API to know what Unpin means, I decided to “fix” the problem upstream and pin on the heap.

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

4. Example 1: $1 \rightarrow 1$ Operator



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>>`:

```
1 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>>`:

```
1 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`:

```

1 fn poll_next(self: Pin<&mut Self>, cx: &mut Context<'_>)
2     -> Poll<Option<Self::Item>> {
3     let this = self.get_mut(); // Safe: Double is Unpin now
4     this.in_stream.as_mut()    // Project to Pin<&mut InSt>
5         .poll_next(cx)
6         .map(|opt| opt.map(|x| x * 2))
7 }
```

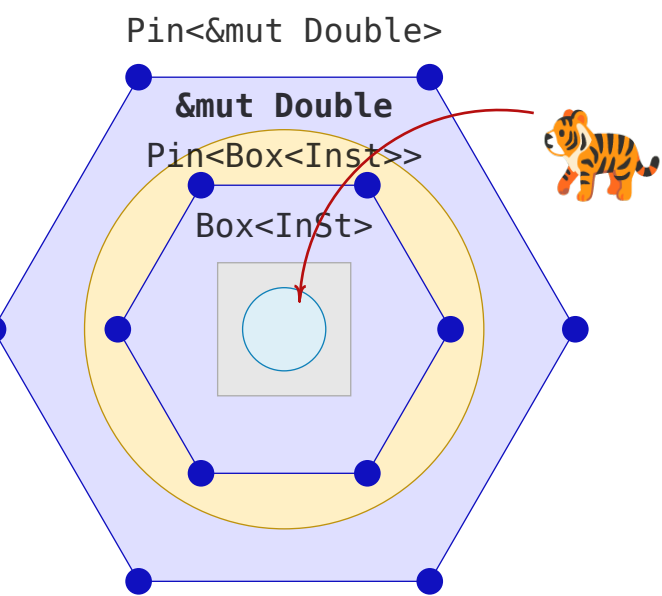


This works **without requiring `InSt: Unpin`**!

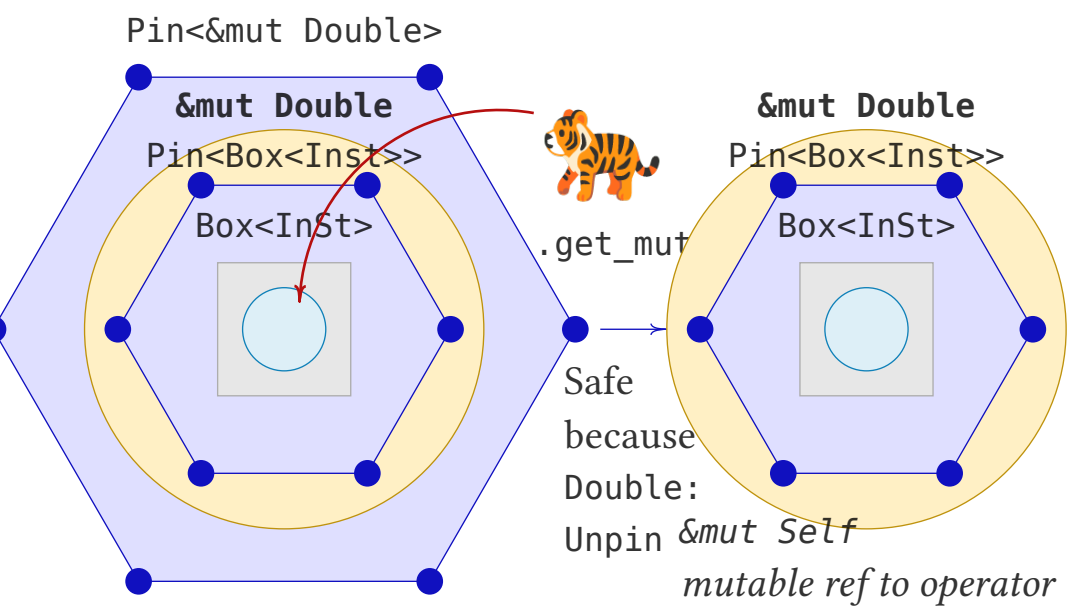
4.7. Turning !Unpin into Unpin with boxing

4. Example 1: 1 → 1 Operator

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



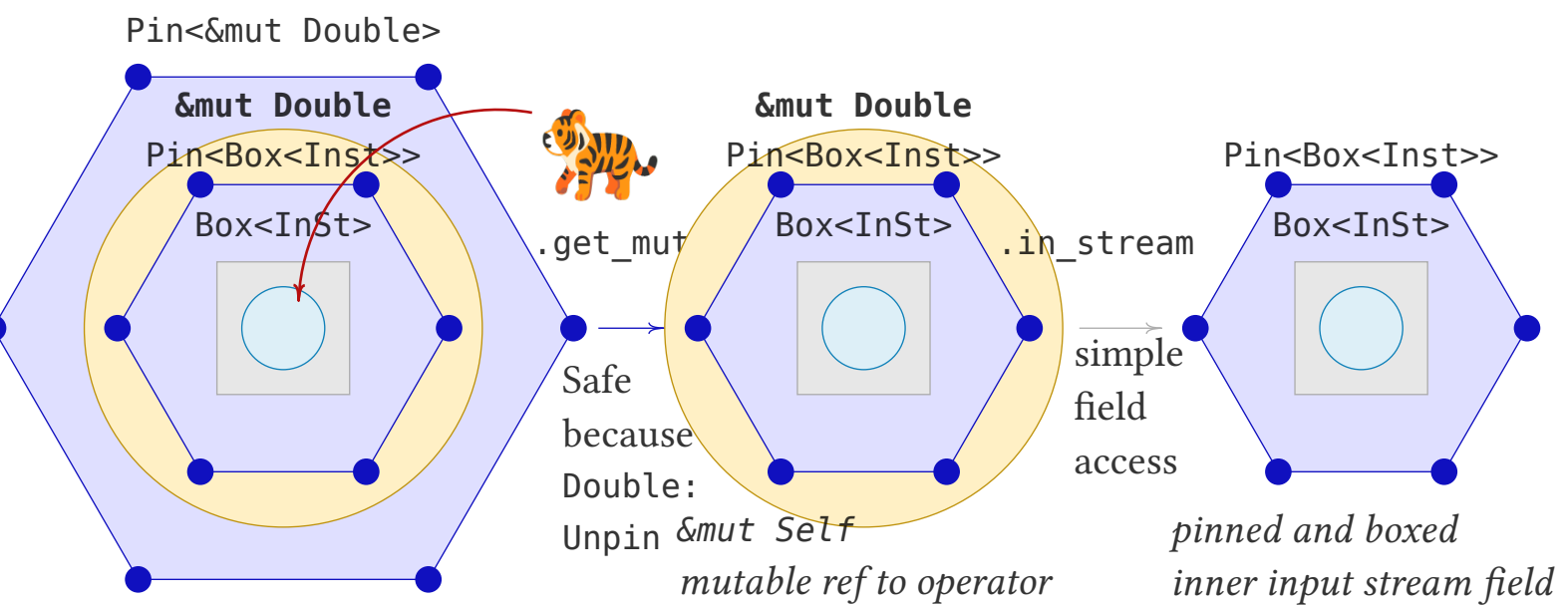
From `Pin<&mut Double>` to `Pin<&mut InSt>` in a few **safe steps**:



4.7. Turning !Unpin into Unpin with boxing

4. Example 1: 1 → 1 Operator

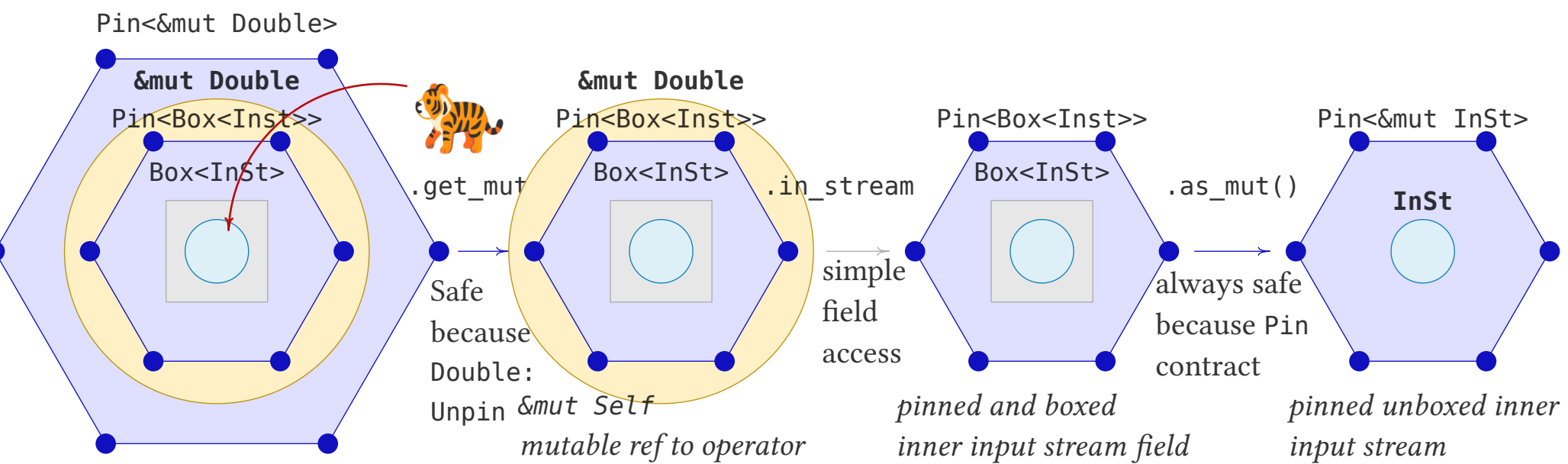
From `Pin<&mut Double>` to `Pin<&mut InSt>` in a few **safe steps**:



4.7. Turning !Unpin into Unpin with boxing

4. Example 1: 1 → 1 Operator

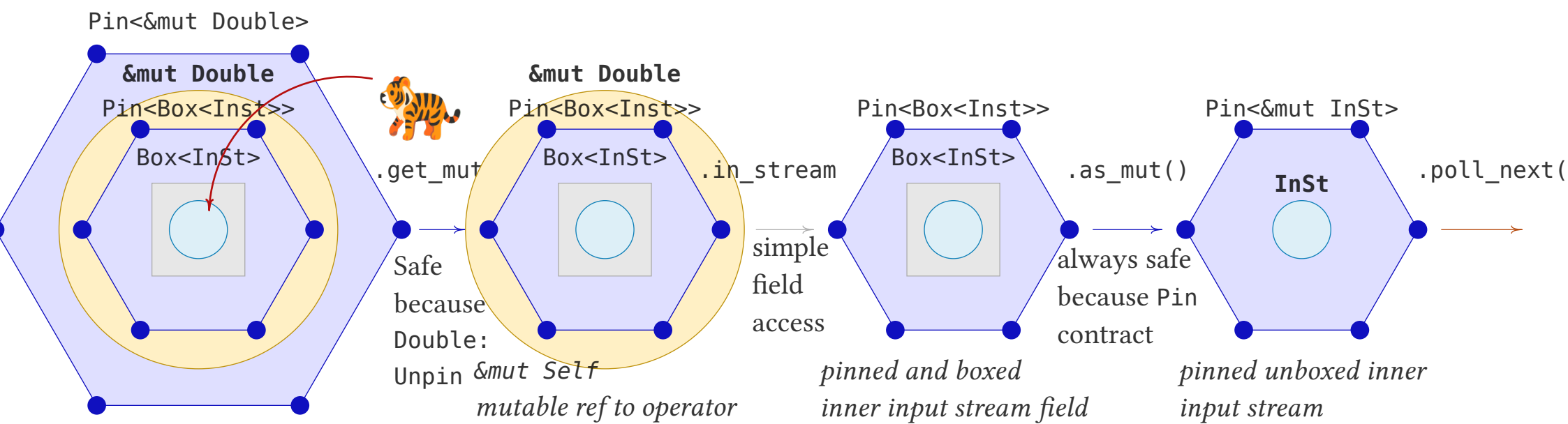
From `Pin<&mut Double>` to `Pin<&mut InSt>` in a few **safe steps**:



4.7. Turning !Unpin into Unpin with boxing

4. Example 1: 1 → 1 Operator

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



4.7. Turning !Unpin into Unpin with boxing

4. Example 1: $1 \rightarrow 1$ Operator

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

```
1  impl<InSt> Stream for Double<InSt>
2  where InSt: Stream<Item = i32>
3  {
4      fn poll_next(self: Pin<&mut Self>, cx: &mut Context<'_>)
5          -> Poll<Option<Self::Item>>
6      {
7          // We can project because `Self: Unpin`
8          let this: &mut Double<InSt> = self.get_mut();
9          this.in_stream.as_mut()
10             .poll_next(cx)
11             .map(|r| r.map(|x| x * 2))
12      }
13 }
```




4.8. Review of approaches to !Unpin fields

4. Example 1: $1 \rightarrow 1$ Operator

Approach 1: Use Box<_>




```
1 struct Double<InSt> {  Rust
2   in_stream: Pin<Box<InSt>>
3 }
4
5 impl<InSt> Stream for
  Double<InSt>
6   where InSt: Stream
```

✓ Works with any InSt, also !Unpin

4.8. Review of approaches to !Unpin fields


4. Example 1: $1 \rightarrow 1$ Operator

Approach 1: Use Box<_>

```
1 struct Double<InSt> {  Rust
2   in_stream: Pin<Box<InSt>>
3 }
4
5 impl<InSt> Stream for
  Double<InSt>
6   where InSt: Stream
```

✓ Works with any InSt, also !Unpin

Approach 2: Require Unpin

```
1 struct Double<InSt> {  Rust
2   in_stream: InSt
3 }
4
5 impl<InSt> Stream for
  Double<InSt>
6   where InSt: Stream + Unpin
```

✗ Imposes Unpin constraint on users

Approach 3: Use pin-project crate



4.8. Review of approaches to !Unpin fields

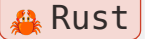
Approach 3: Projection with pin-project

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

```

1  #[pin_project]
2  struct Double<InSt> {
3      #[pin]
4      in_stream: InSt,
5  }
6  impl<InSt: Stream> Stream for Double<InSt> {
7      fn poll_next(self: Pin<&mut Self>, cx: &mut Context<'_>)
8          -> Poll<Option<Self::Item>>
9      {
10         //
11         self.project().in_stream.poll_next(cx)
12             .map(|r| r.map(|x| x * 2))
13     }
14 }

```



Info

pin-project generates a safe projection method `project()`.
You don't have to juggle with Unpin (**but your users have to!**)

4.9. Distributing your operator

4. Example 1: $1 \rightarrow 1$ Operator

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



Rust

```
1 trait DoubleStream: Stream {  
2     fn double(self) -> Double<Self>  
3     where Self: Sized + Stream<Item = i32>,  
4     { Double::new(self) }  
5 }  
  
6 // A blanket implementation should be provided by you!  
7 impl<S> DoubleStream for S where S: Stream<Item = i32> {}
```


4.9. Distributing your operator

4. Example 1: $1 \rightarrow 1$ Operator

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



```
1 trait DoubleStream: Stream {  
2     fn double(self) -> Double<Self>  
3     where Self: Sized + Stream<Item = i32>,  
4     { Double::new(self) }  
5 }  
  
6 // A blanket implementation should be provided by you!  
7 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



Stream Trait Interface

Lazy: `.poll_next()` only responds when called



Data pushed up

Leaf Streams (Real Drivers)

TCP, Files, Timers, Hardware, Channels

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

1.	Introduction	1
2.	Rust's Stream trait	8
3.	Using Streams	14
4.	Example 1: $1 \rightarrow 1$ Operator	20
5.	Example 2: $1 \rightarrow N$ Operator	44
5.1.	Complexity $1 \rightarrow N$ operators	45
5.2.	Sharing latency between tasks	46
5.3.	Handling sleeping and waking	48
5.4.	Simplified state machine of clone-stream	49
6.	Conclusion	50

5.1. Complexity $1 \rightarrow N$ operators

5. Example 2: $1 \rightarrow N$ Operator

Challenges for Stream operators are combined from:

Inherent Future challenges:

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



5.1. Complexity $1 \rightarrow N$ operators

5. Example 2: $1 \rightarrow N$ Operator

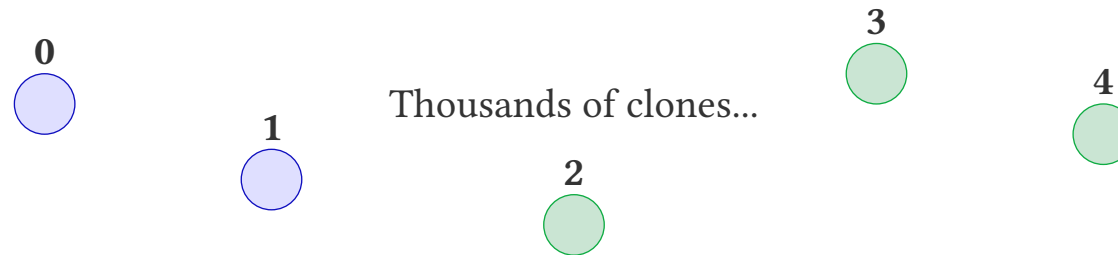
Challenges for Stream operators are combined from:

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

5. Example 2: $1 \rightarrow N$ Operator

Latency may need to be processed by different async tasks:



```
1 let tcp_stream =  
  TcpStream::connect("127.0.0.1:8080").await?;  
2 let latency = tcp_stream.latency(); // Stream<Item = Duration>  
3  
4 spawn(async move { display_ui(latency).await; });  
5 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!

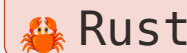
5.2. Sharing latency between tasks

5. Example 2: $1 \rightarrow N$ Operator

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



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



```
2
```

```
3 let breaks_latency_clone = ui_latency.clone();
```

```
4 // Warning: `Clone` needs to be implemented!
```

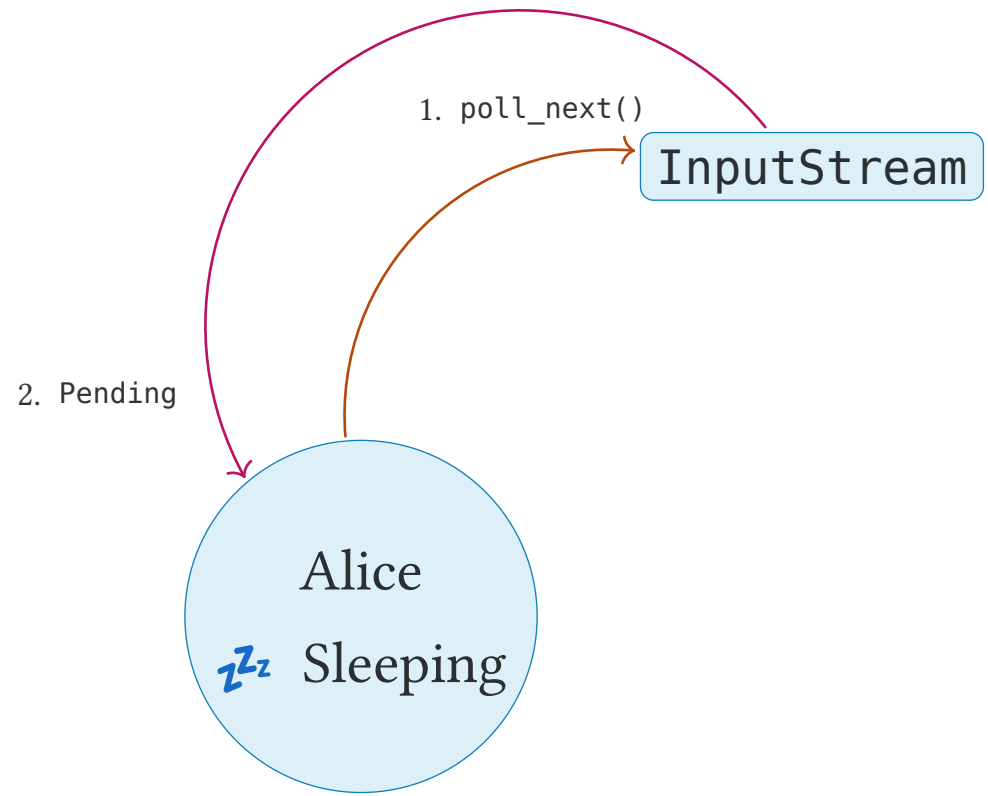
```
5
```

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

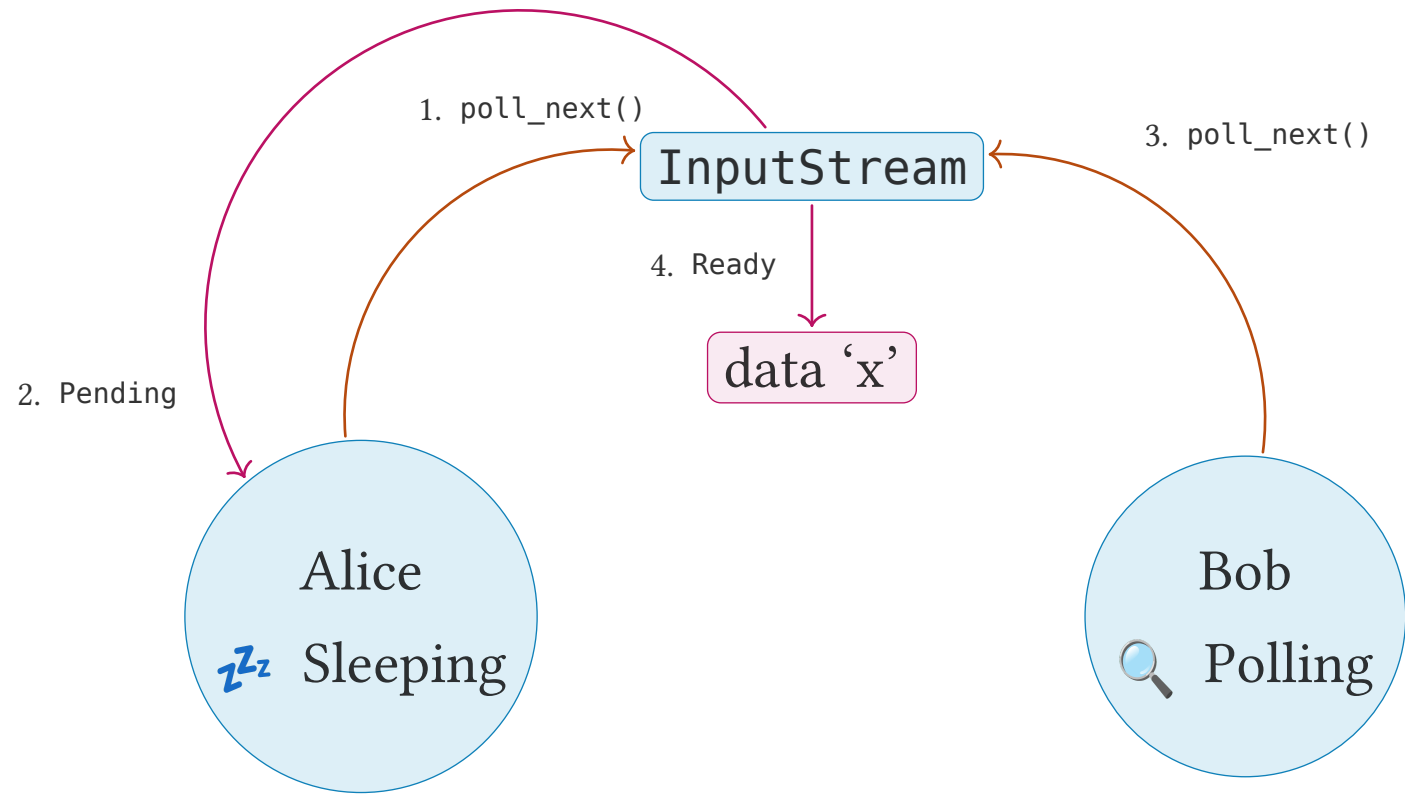
```
7 spawn(async move  
  { engage_breaks(breaks_latency_clone).await; });
```

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

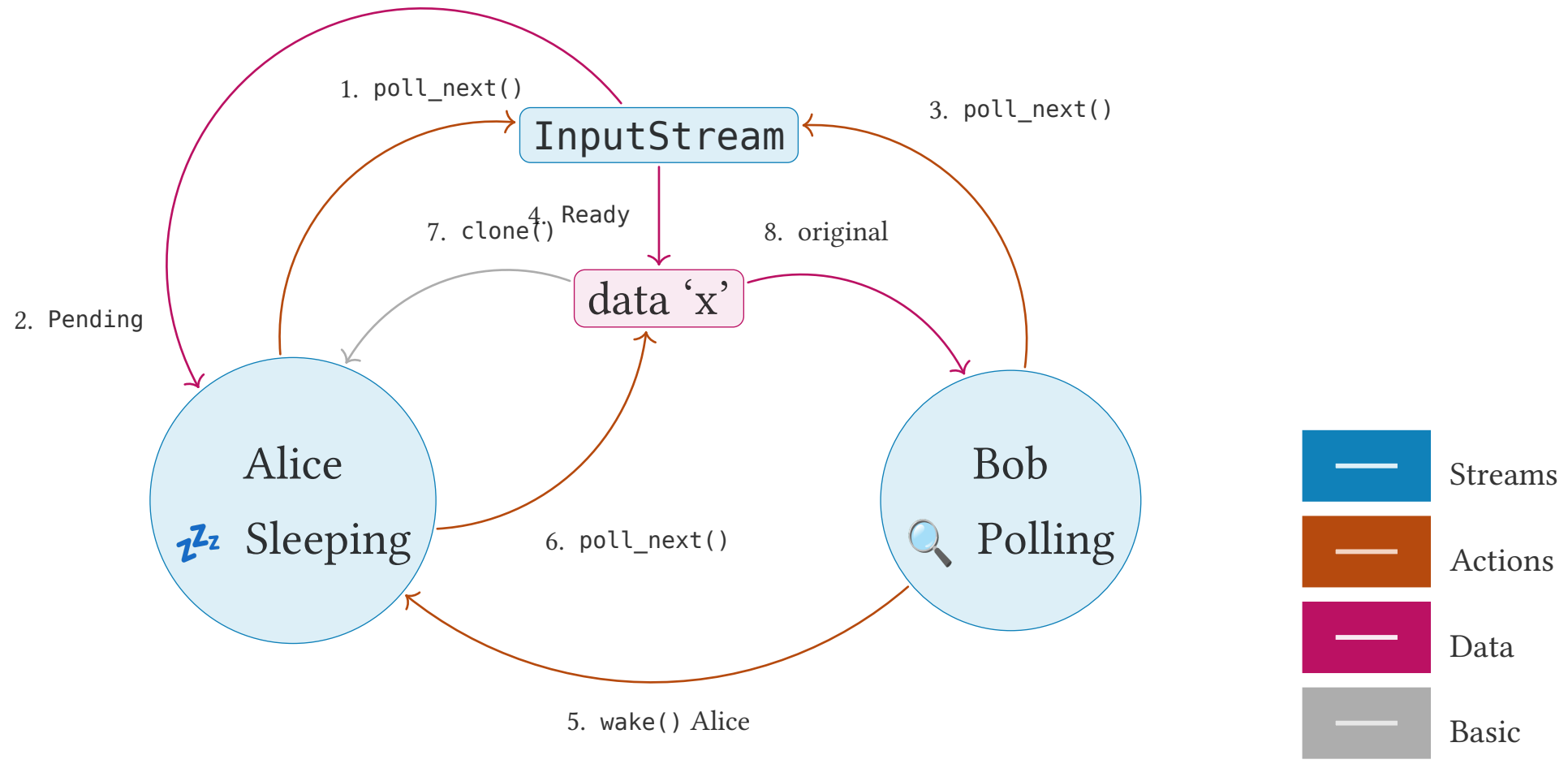
5.3. Handling sleeping and waking



5.3. Handling sleeping and waking



5.3. Handling sleeping and waking



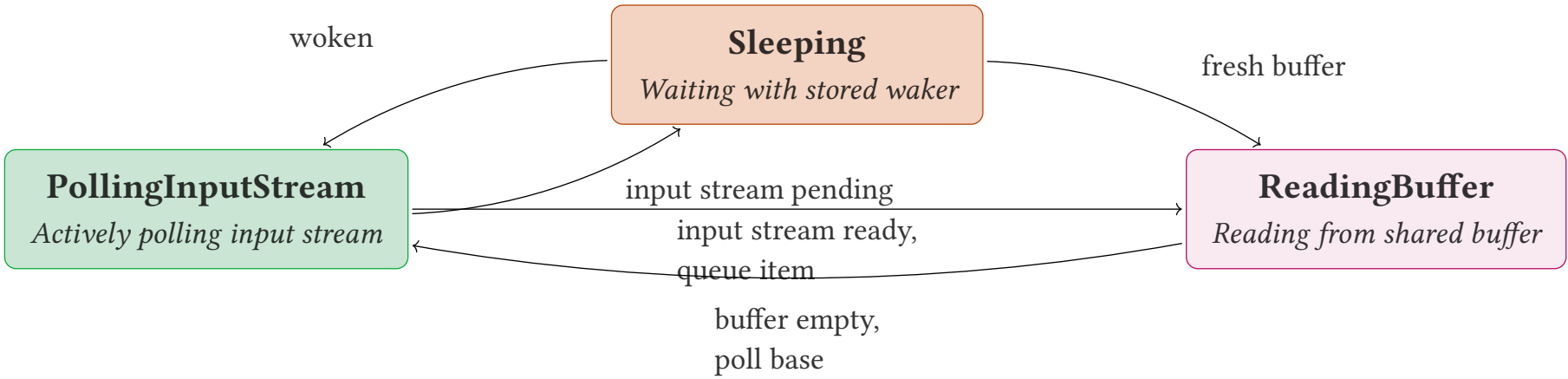
5.4. Simplified state machine of clone-stream

5. Example 2: $1 \rightarrow N$ Operator

Enforcing simplicity, correctness and performance:



Warning
Each clone maintains its own state



5.4. Simplified state machine of clone-stream

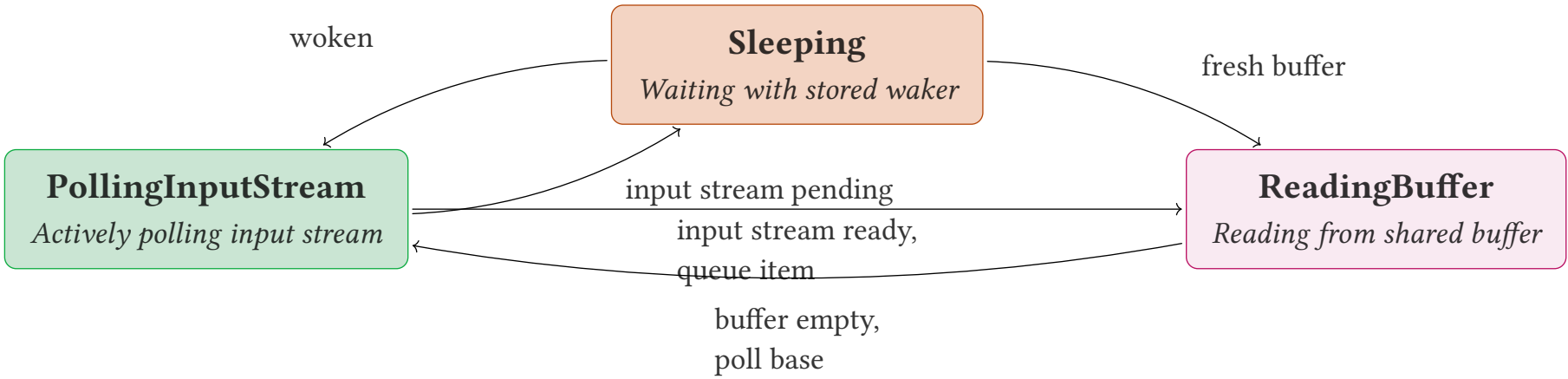
5. Example 2: $1 \rightarrow N$ Operator

Enforcing simplicity, **correctness** and **performance**:



Warning

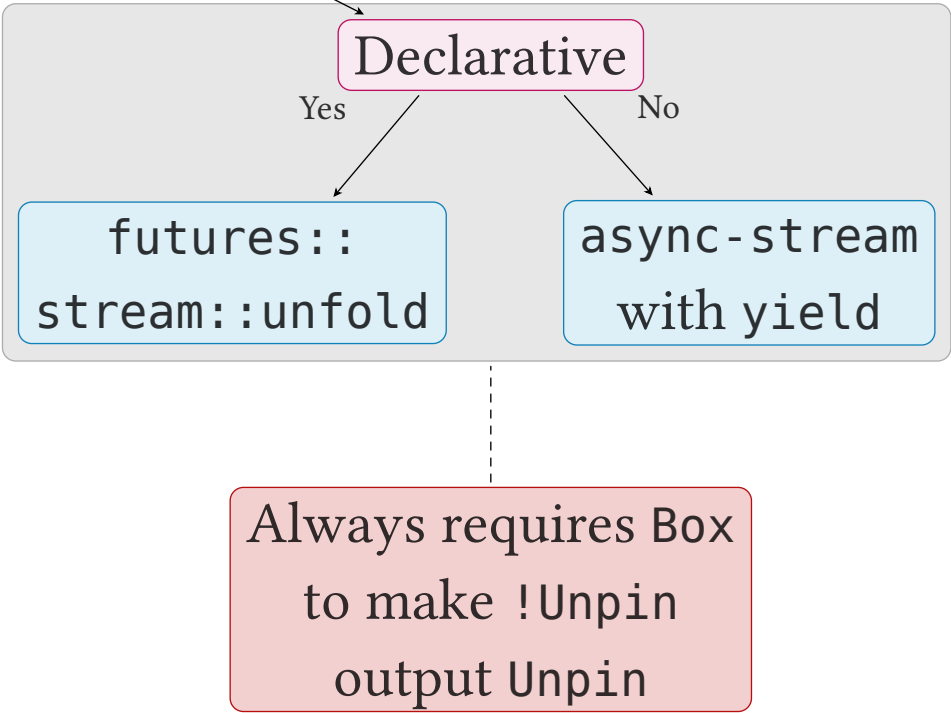
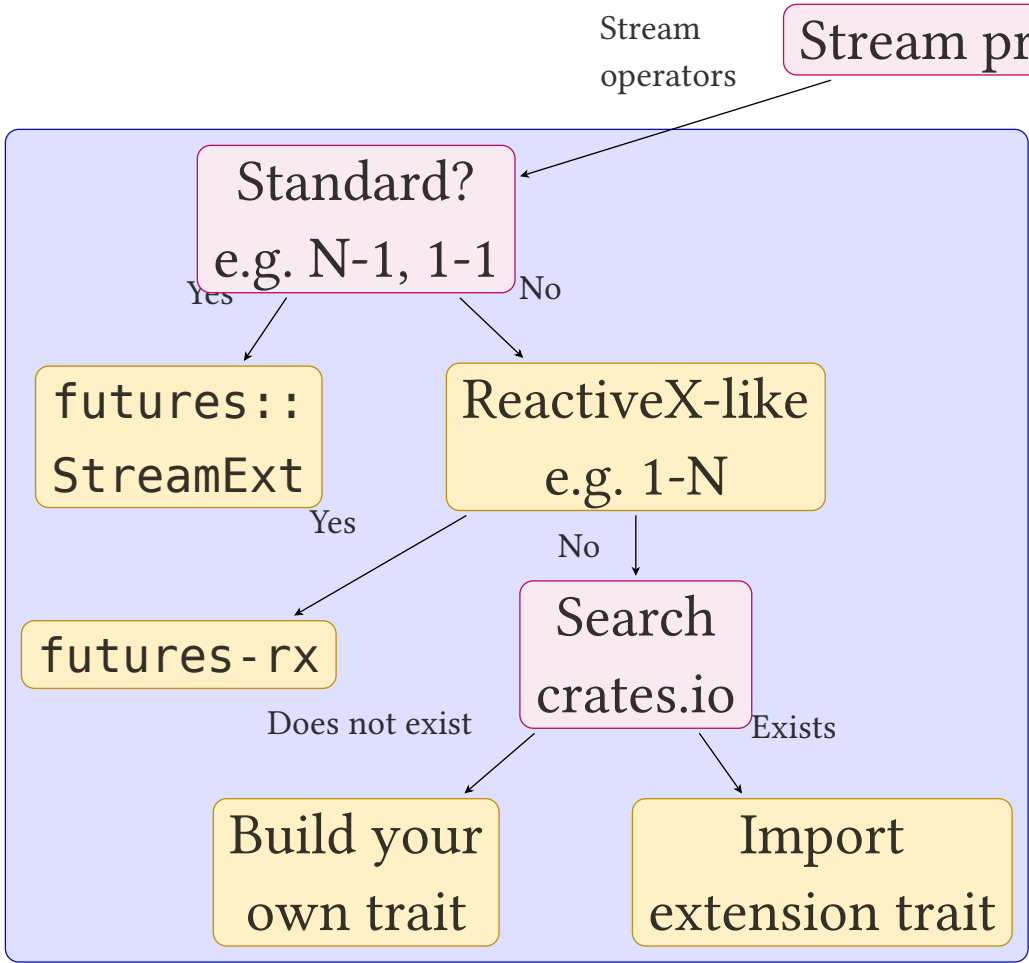
Each clone maintains its own state

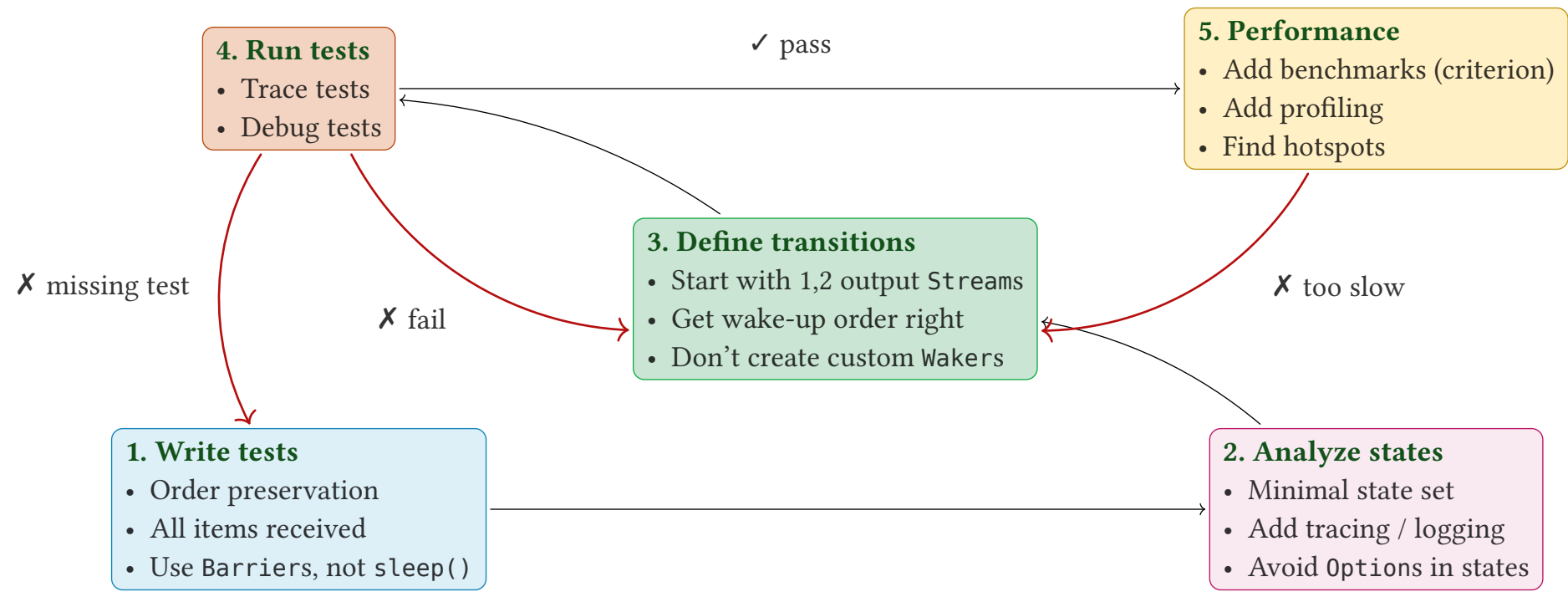


Speed

8 - 12 micro seconds per item per clone. (Using pin-project slowed down.)

1. Introduction	1
2. Rust's Stream trait	8
3. Using Streams	14
4. Example 1: $1 \rightarrow 1$ Operator	20
5. Example 2: $1 \rightarrow N$ Operator	44
6. Conclusion	50
6.1. Quickstart	51
6.2. Advanced operator construction	52
6.3. Questions	53





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/