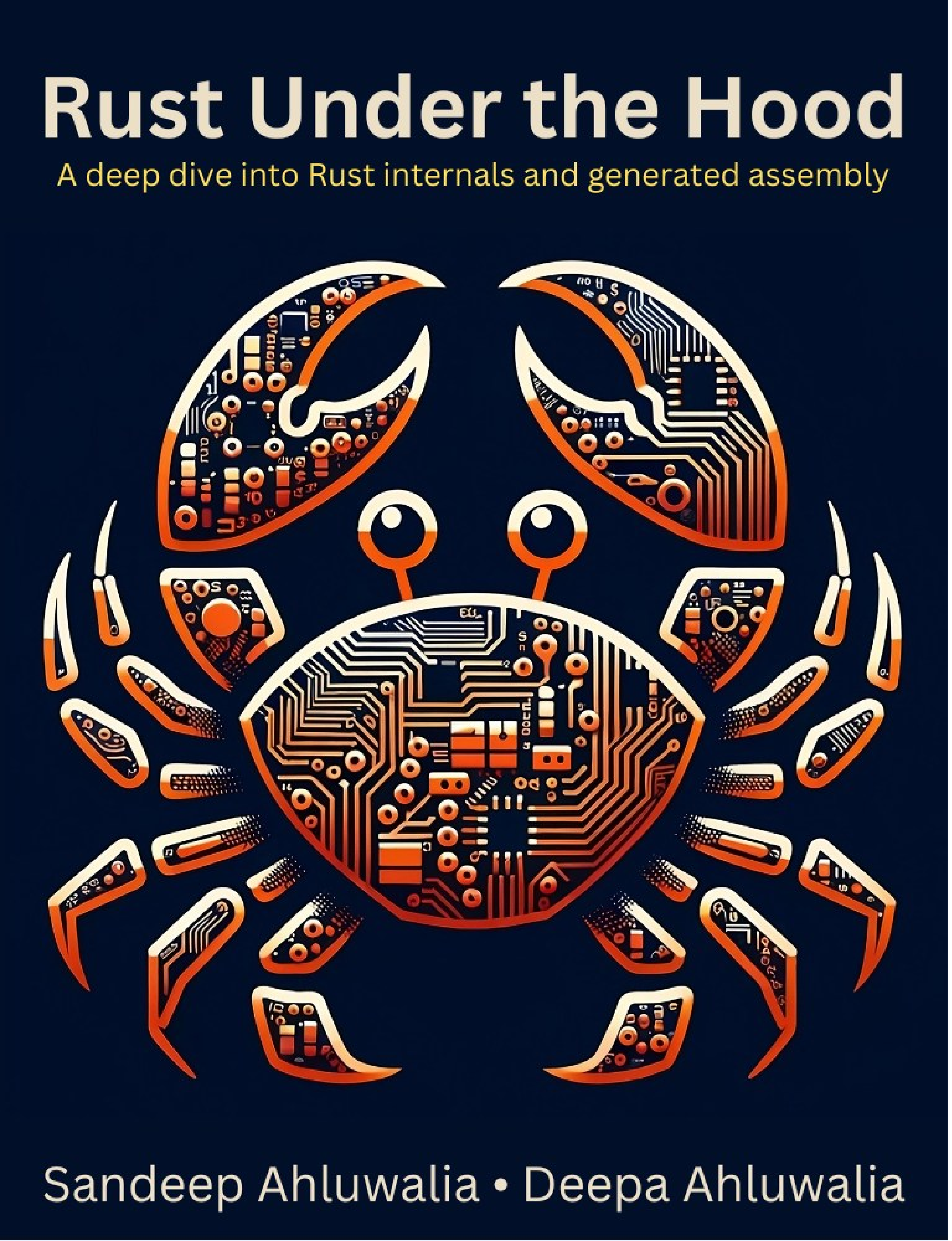


Rust Under the Hood

Sandeep Ahluwalia
Deepa Ahluwalia

https://eventhelix.com

Rust Under the Hood

- Memory layout of enums, struct, vectors, strings, and arrays
- Pattern matching internals
- How smart pointers manage memory
- Tail call optimization and recursion
- Dynamic dispatch and vtables
- Functional programming is a zero-cost abstraction
- How closures capture the environment
- How async/await desugars into futures and state machines

Memory layout of **Number**

```rust
pub enum Number {
    Integer(i64),
    Float(f64),
    Complex { real: f64, imaginary: f64 },
}
```

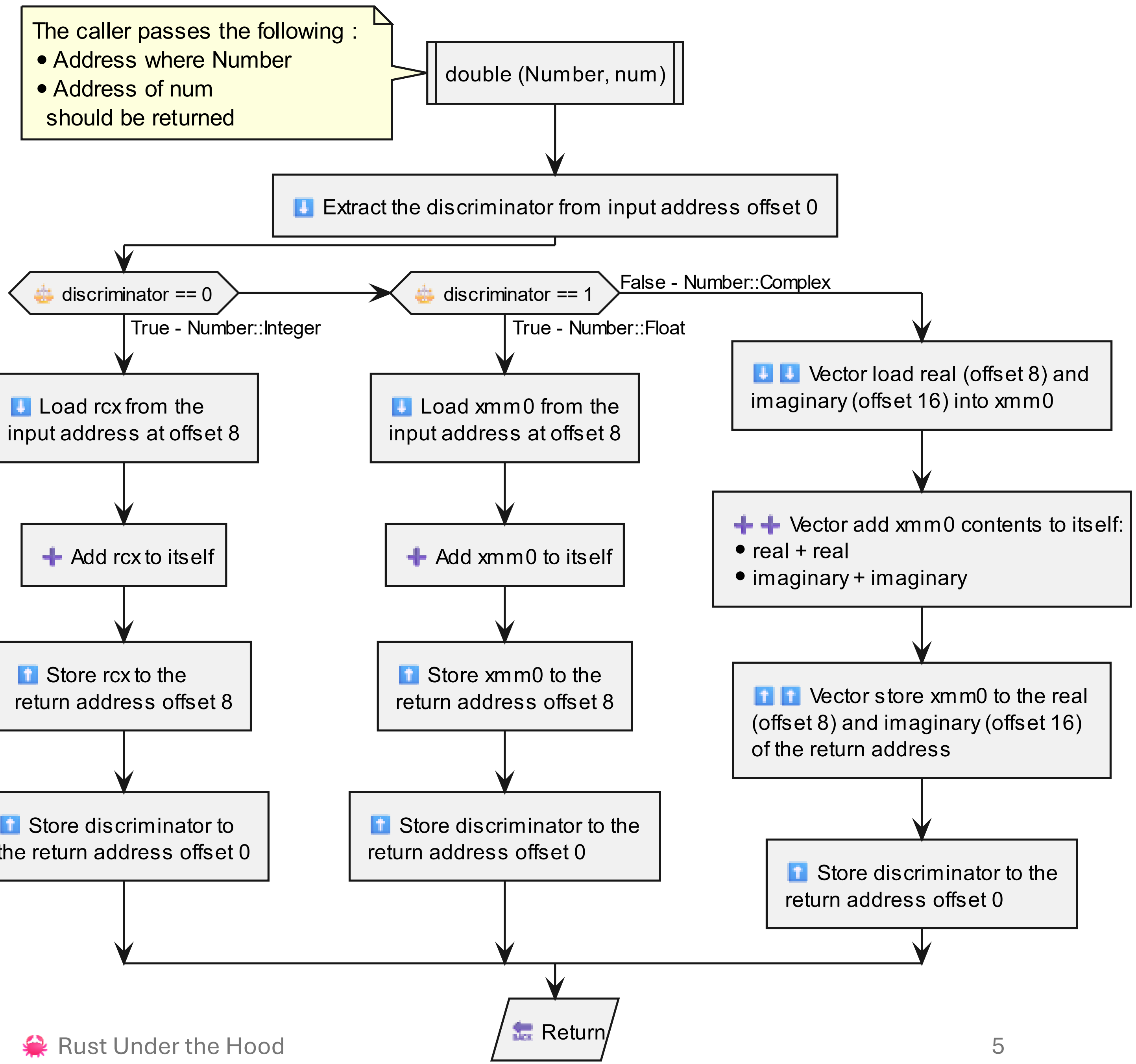
Byte offset	Integer	Float	Complex
0	Discriminator (0)	Discriminator (1)	Discriminator (2)
8	**i64**	**f64**	**f64**
16			**f64**

double: Example of pattern matching

```rust
pub fn double(num: Number) -> Number {
    match num {
        Number::Integer(n) => Number::Integer(n + n),
        Number::Float(n) => Number::Float(n + n),
        Number::Complex { real, imaginary } => Number::Complex {
            real: real + real,
            imaginary: imaginary + imaginary,
        },
    }
}
```

Flow chart of generated assembly of **double**

```
pub fn double(num: Number) -> Number {
    match num {
        Number::Integer(n) => Number::Integer(n + n),
        Number::Float(n) => Number::Float(n + n),
        Number::Complex { real, imaginary } => Number::Complex {
            real: real + real,
            imaginary: imaginary + imaginary,
        },
    }
}
```

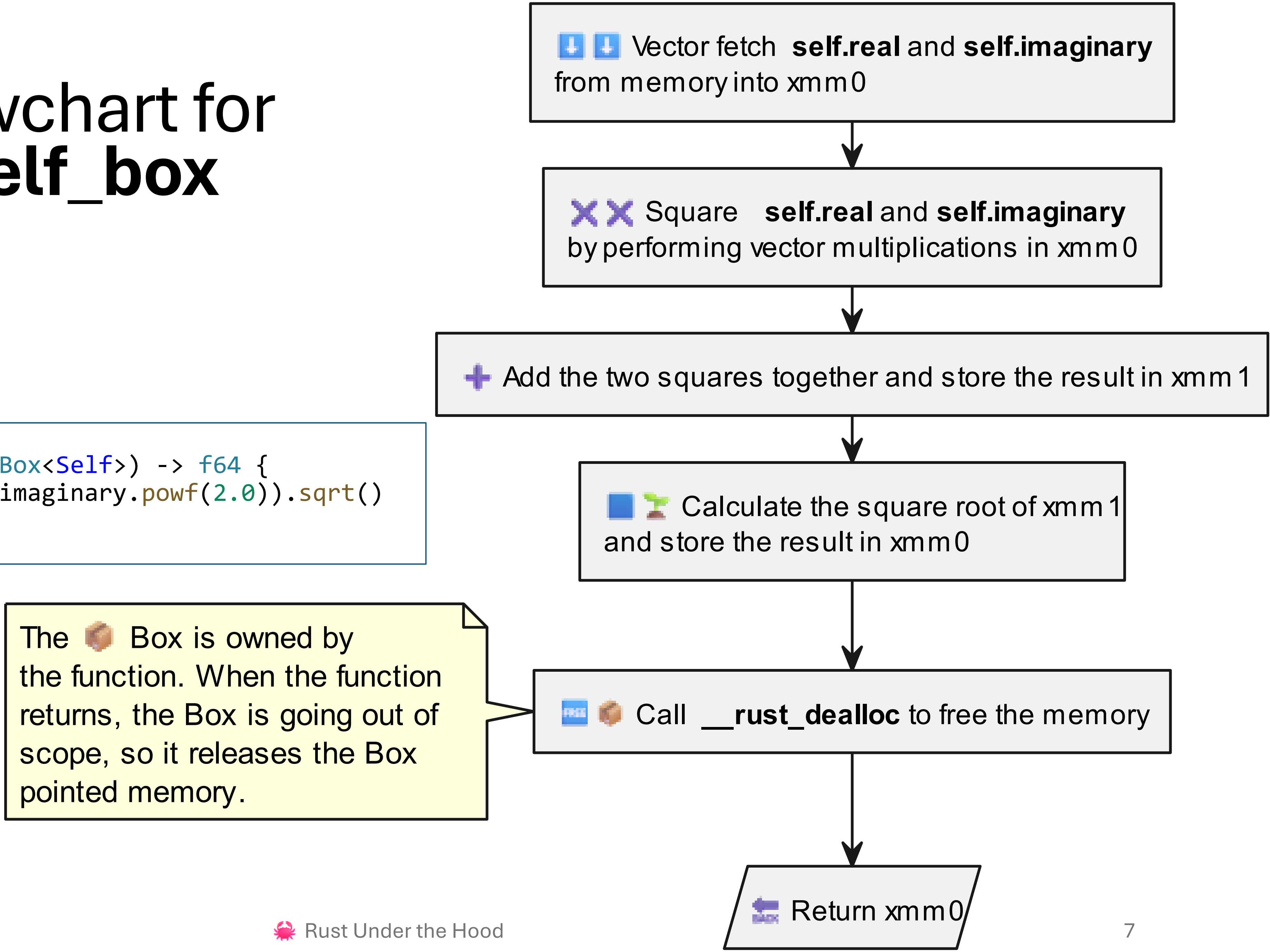


Passing **self** as **Box** and **Arc**

```rust
use std::rc::Rc;
use std::sync::Arc;

#[derive(Copy, Clone)]
pub struct Complex {
    real: f64,
    imaginary: f64,
}

impl Complex {
    // Passing smart pointers
    pub fn magnitude_self_box(self: Box<Self>) -> f64 {
        (self.real.powf(2.0) + self.imaginary.powf(2.0)).sqrt()
    }

    pub fn magnitude_self_arc(self: Arc<Self>) -> f64 {
        (self.real.powf(2.0) + self.imaginary.powf(2.0)).sqrt()
    }
}
```

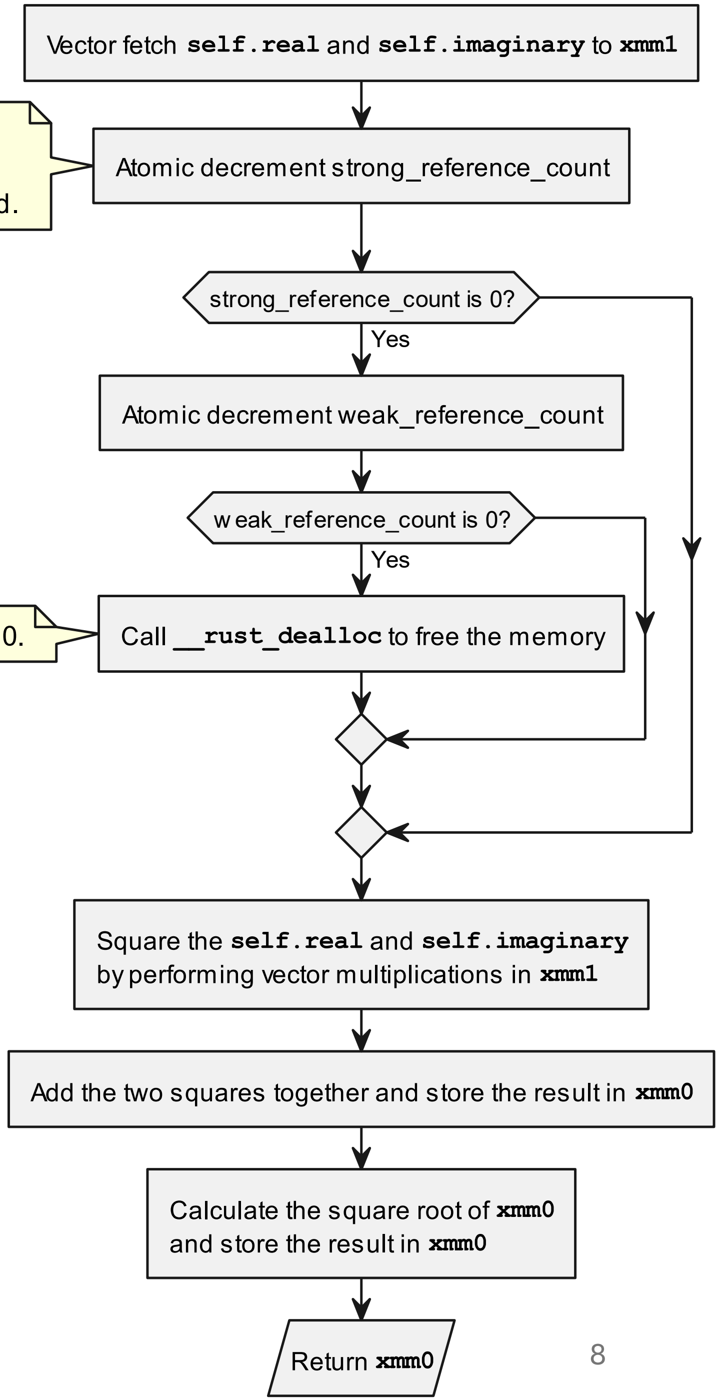
Assembly flowchart for **magnitude_self_box**

```rust
pub fn magnitude_self_box(self: Box<Self>) -> f64 {
    (self.real.powf(2.0) + self.imaginary.powf(2.0)).sqrt()
 }
```



Assembly flowchart for **magnitude_self_arc**

```rust
pub fn magnitude_self_arc(self: Arc<Self>) -> f64 {
      (self.real.powf(2.0) + self.imaginary.powf(2.0)).sqrt()
}
```



Rust **struct** memory layout

```
pub struct MyStruct {
    a: u8,
    b: u64,
    c: i8,
    d: i64,
    e: i32,
}
```

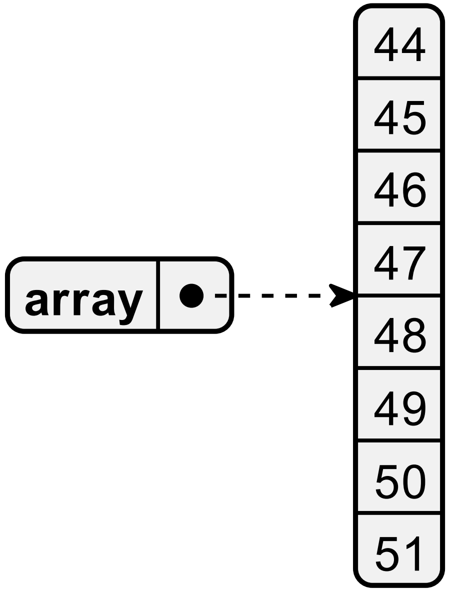
00 | b	u64
08 | d	i64
16 | e	i32
20 | a	u8
21 | c	i8
22 |	2-byte padding

C-compatible Rust **struct** memory layout

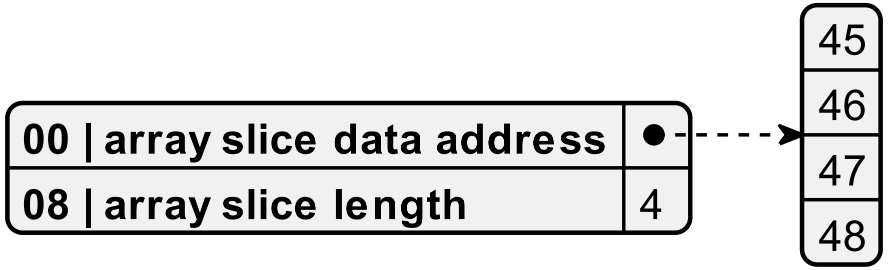
```
#[repr(C)]
pub struct MyStruct {
    a: u8,
    b: u64,
    c: i8,
    d: i64,
    e: i32,
}
```

00 | a	u8
01 |	7-byte padding
08 | b	u64
16 | c	i8
17 |	7-byte padding
24 | d	i64
32 | e	i32

Array



Array slice

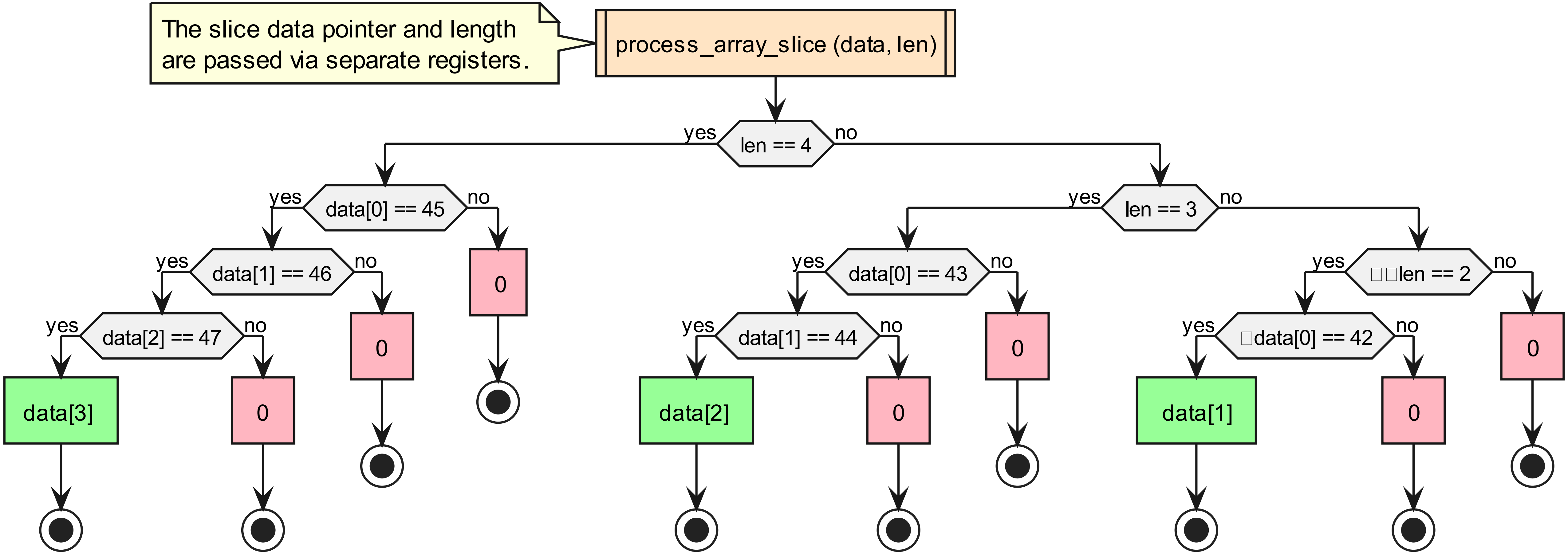


Array Slice Example

```rust
pub fn process_array_slice(input: &[i32]) -> i32 {
    match input {
        [42, a] => *a,
        [43, 44, a] => *a,
        [45, 46, 47, a] => *a,
        [..] => 0,
    }
}
```

Assembly flowchart for **process_array_slice**

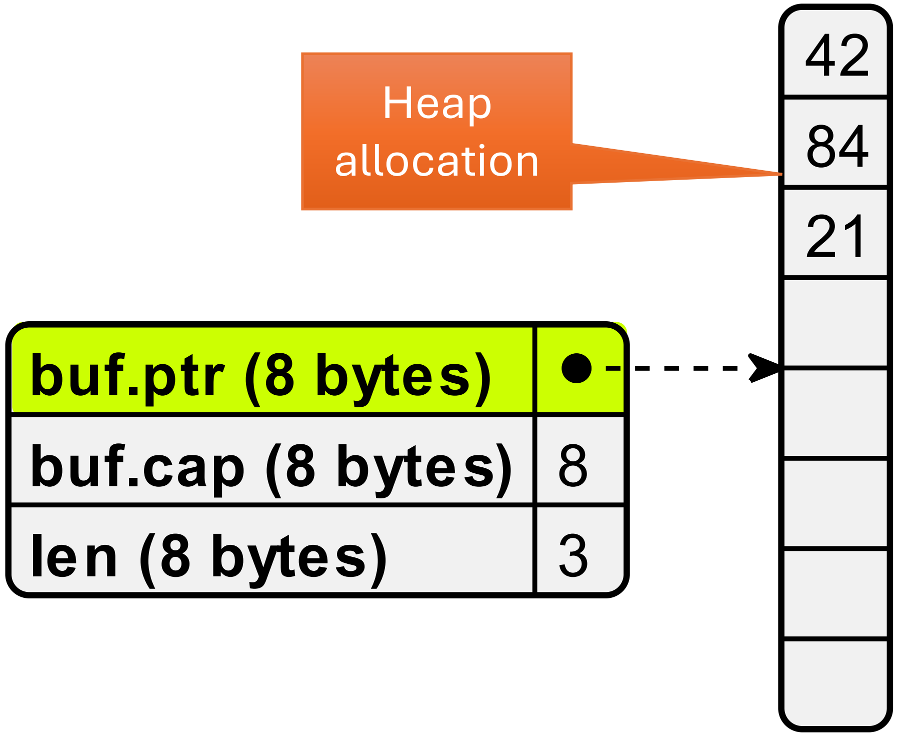
```rust
pub fn process_array_slice(input: &[i32]) -> i32 {
    match input {
        [42, a] => *a,
        [43, 44, a] => *a,
        [45, 46, 47, a] => *a,
        [..] => 0,
    }
}
```



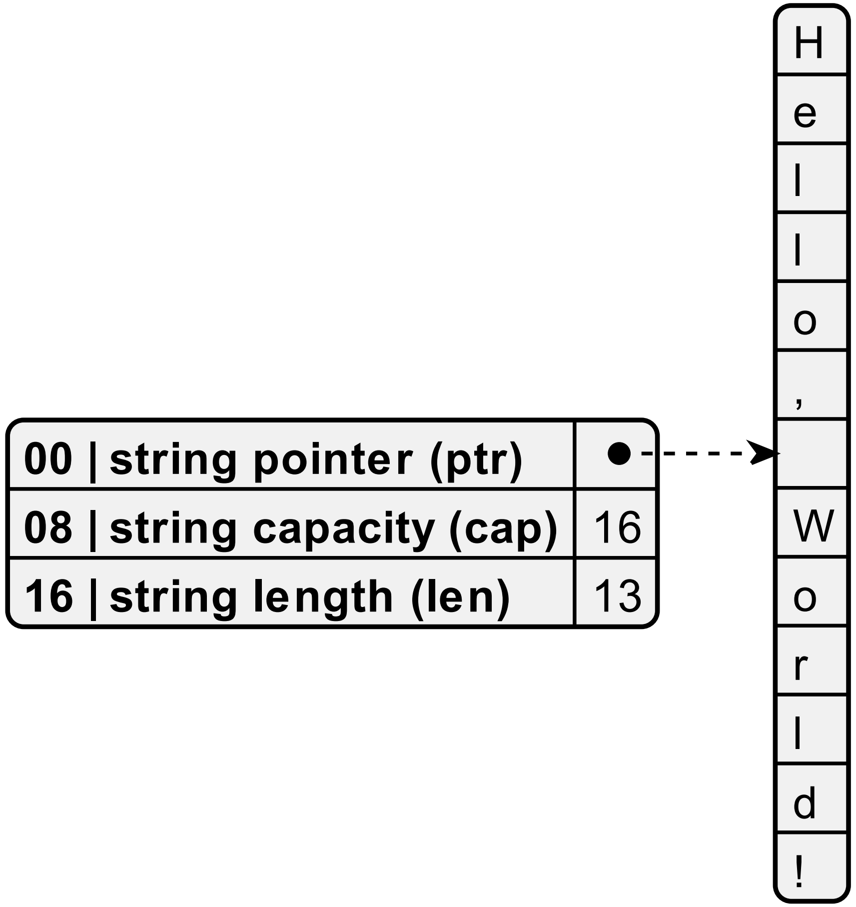
Vec layout

```
pub struct Vec<T, A: Allocator = Global> {
    // Data pointer and capacity
    buf: RawVec<T, A>,
    // Length of the vector
    len: usize,
}
```

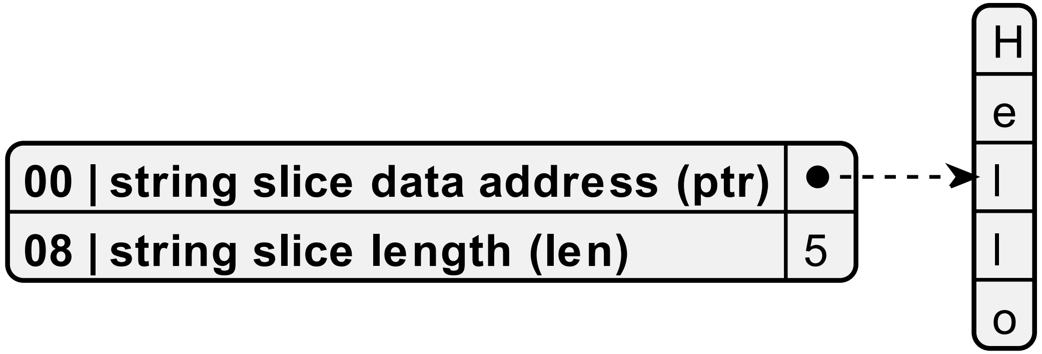
```
struct RawVec<T> {
    // Points to the heap address that
    // stores the vector data
    ptr: NonNull<T>,
    // Capacity of the vector
    cap: usize,
    _marker: PhantomData<T>,
}
```



String layout



String slice layout



Functional programming in Rust is a 0-cost abstraction

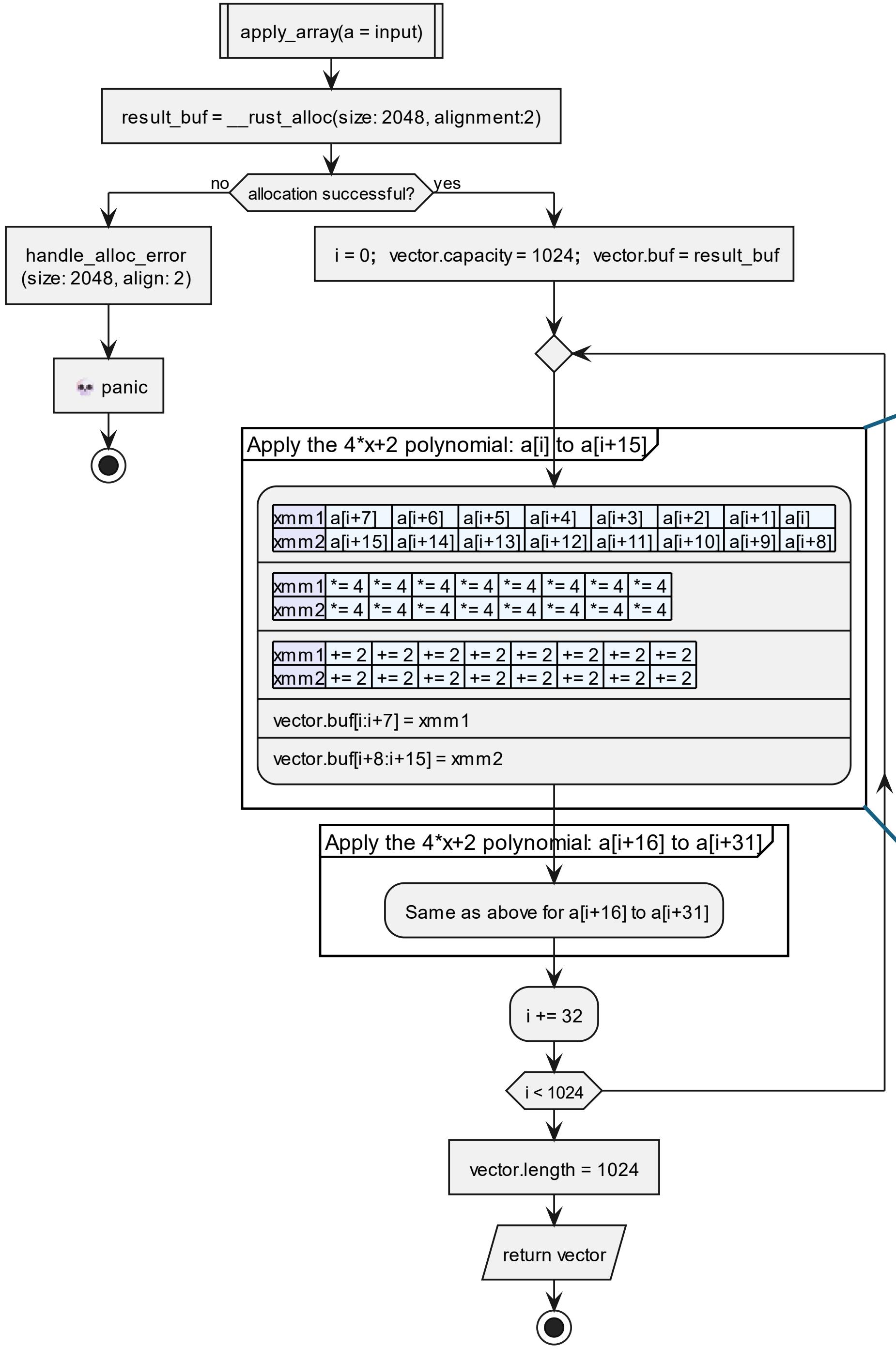
```
type GenFunction<T> = fn(T) -> T;

fn apply_array<T: Copy, const N: usize>(
    input: &[T],
    functions: &[GenFunction<T>; N],
) -> Vec<T> {
    input
        .iter()
        .map(|&item| functions.iter().fold(item, |acc, &f| f(acc)))
        .collect()
}
```

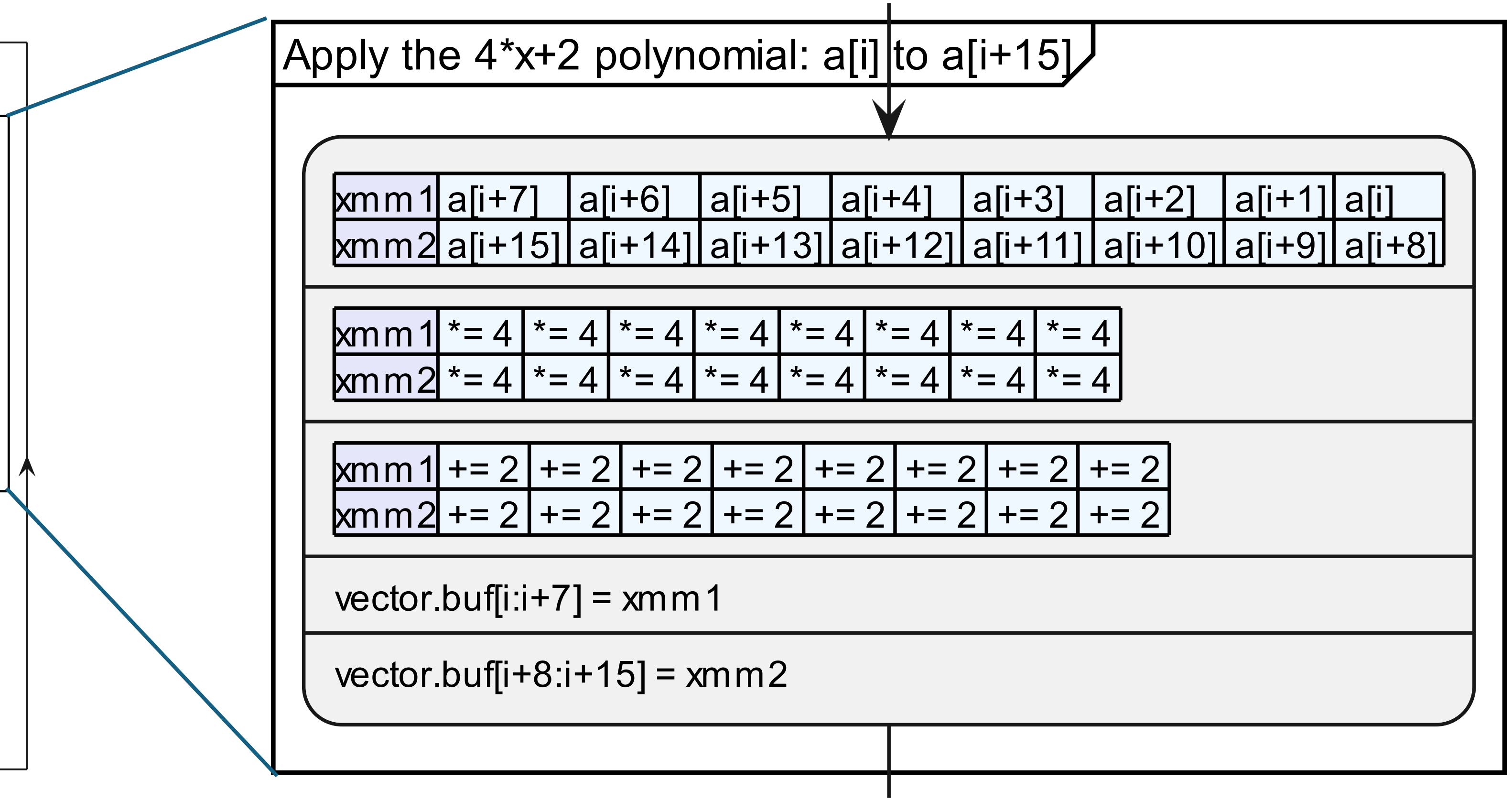
```rust
type Num = i16;
const ARRAY_SIZE: usize = 1024;

fn add_one(n: Num) -> Num { n + 1 }
fn multiply_by_two(n: Num) -> Num { n * 2 }

const FUNCTIONS_3: [GenFunction<Num>; 3] =
              [multiply_by_two, add_one, multiply_by_two];

pub fn apply_array(input: &[Num; ARRAY_SIZE]) -> Vec<Num> {
    apply_functions(input, &FUNCTIONS_3)
}
```

The compiler inlines the three functions into a single polynomial

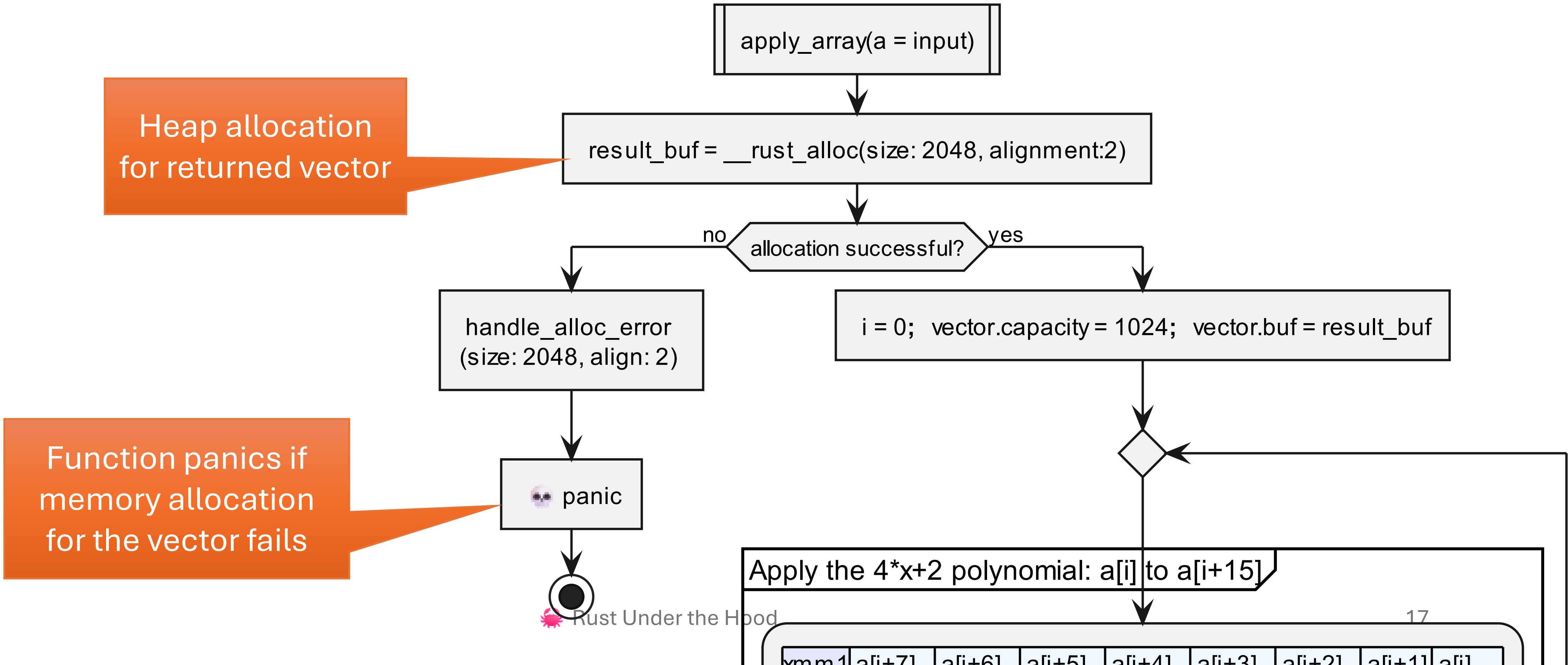
Function	Polynomial	Folded Polynomial
`multiply_by_two`	2x	2x
`add_one`	x + 1	2x + 1
`multiply_by_two`	2x	4x + 2

Assembly flowchart of **apply_array**





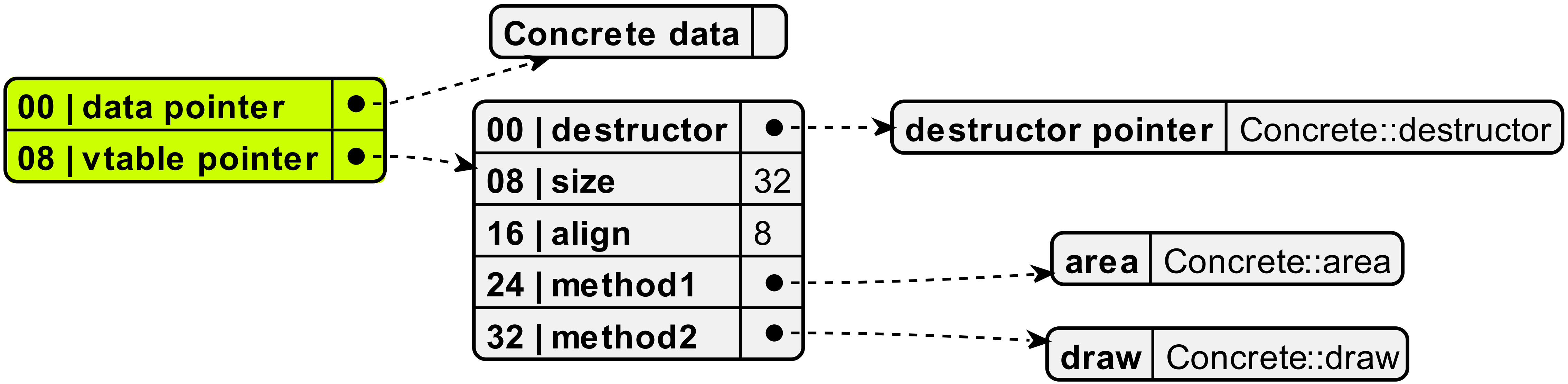
```rust
fn apply_array<T: Copy, const N: usize>(input: &[T],functions: &[GenFunction<T>; N]) -> Vec<T>
{
    input
        .iter()
        .map(|&item| functions.iter().fold(item, |acc, &f| f(acc)))
        .collect()
}
```



Traits, Vtables and Tail Calls

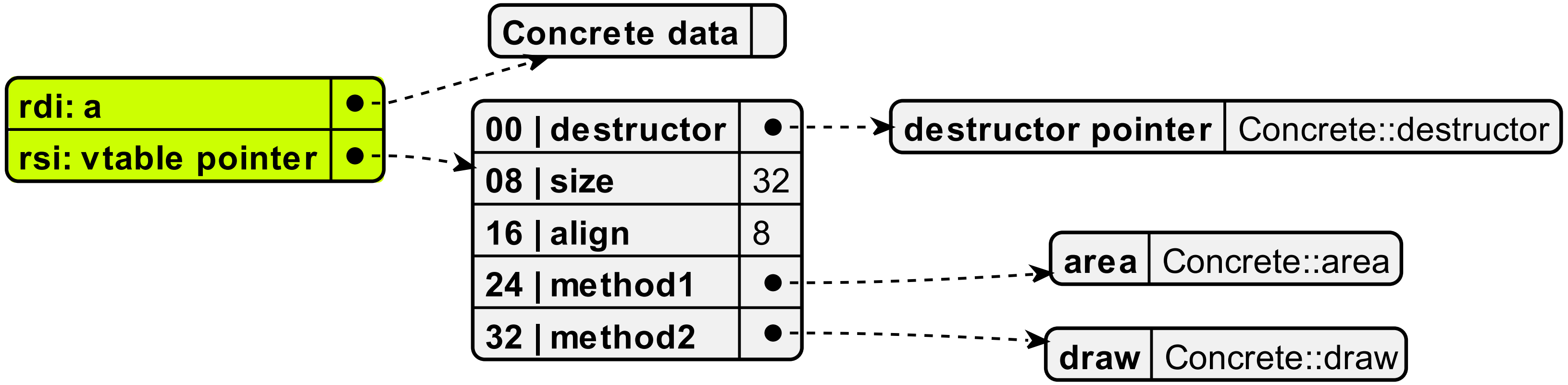
```rust
type Num = f64;

pub trait Shape {
    type T;
    fn area(&self) -> Self::T;
}

pub trait Draw: Shape {
    fn draw(&self);
}

pub fn draw_dynamic(a: &dyn Draw<T = Num>) {
    a.draw();
}

pub fn draw_and_report_area_dynamic(a: &dyn Draw<T = Num>) -> Num {
    a.draw();
    a.area()
}
```

Dynamic dispatch via fat pointers



- Concrete-type methods are referenced via a pointer
- Size, alignment, and destructor are needed for freeing concrete types via the fat-pointer
  - E.g. Box containing a fat pointer is dropped

```rust
pub fn draw_dynamic(a: &dyn Draw<T = Num>) {
    a.draw();
}
```

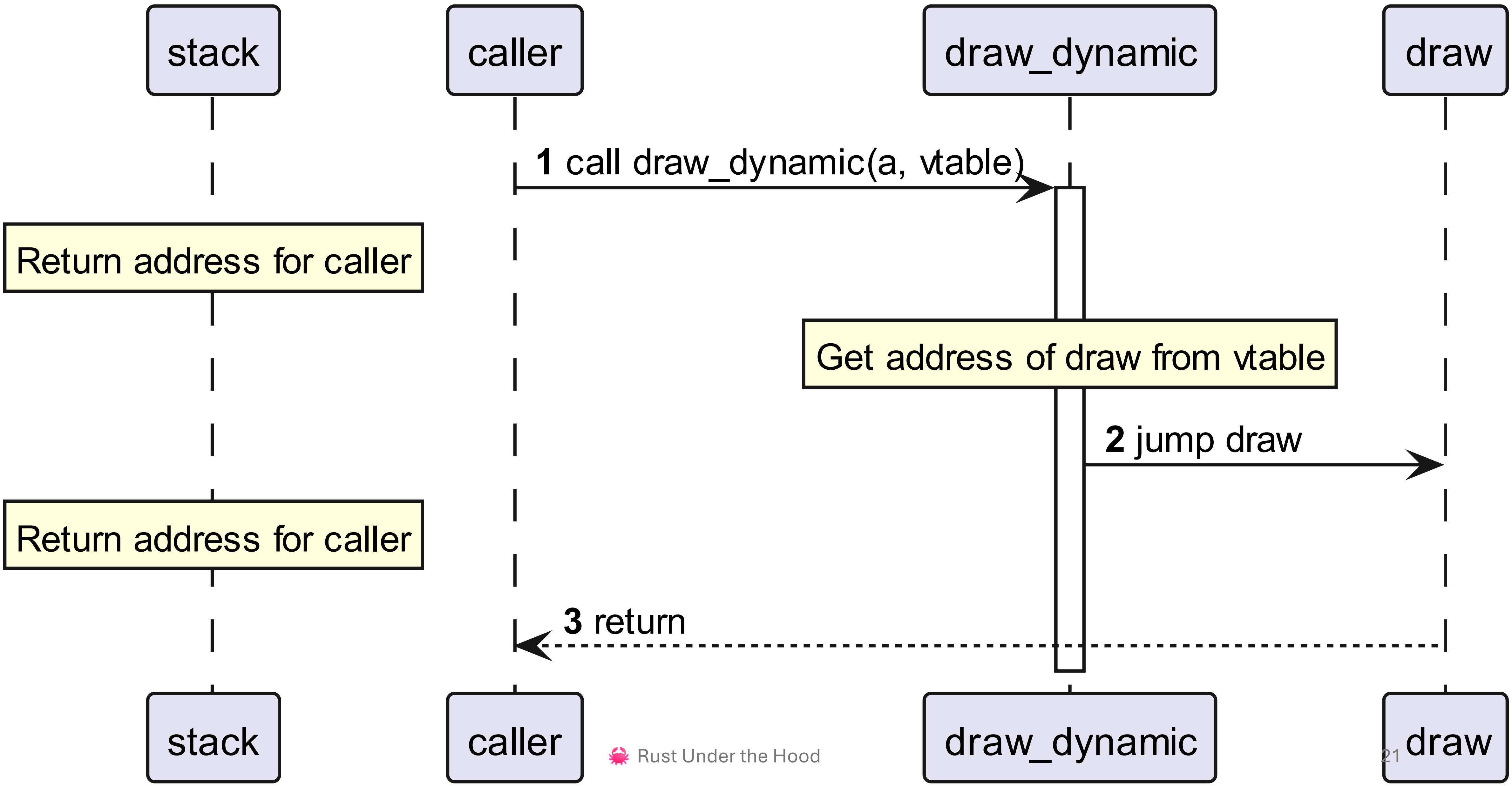



```
example::draw_dynamic:
jmp     qword ptr [rsi + 32] ; 🪱 a.draw() is called via vtable
                             ; tail call optimization is applied.
```

Tail call optimization:

- jmp instead of call
- rdi already contains a

Jump to `draw` (tail call optimization)



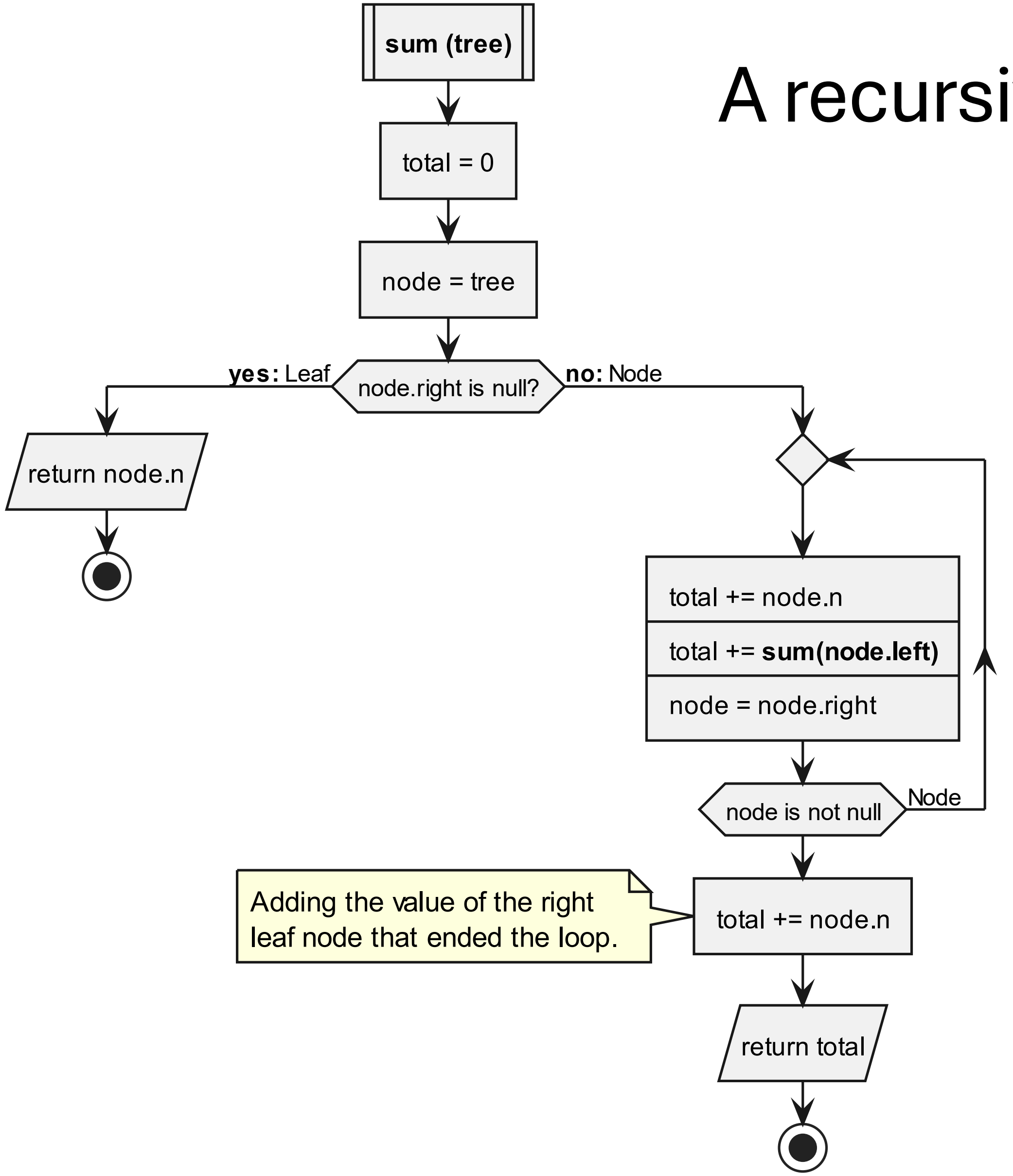
```rust
pub fn draw_and_report_area_dynamic(a: &dyn Draw<T = Num>) -> Num {
    a.draw();
    a.area()
}
```

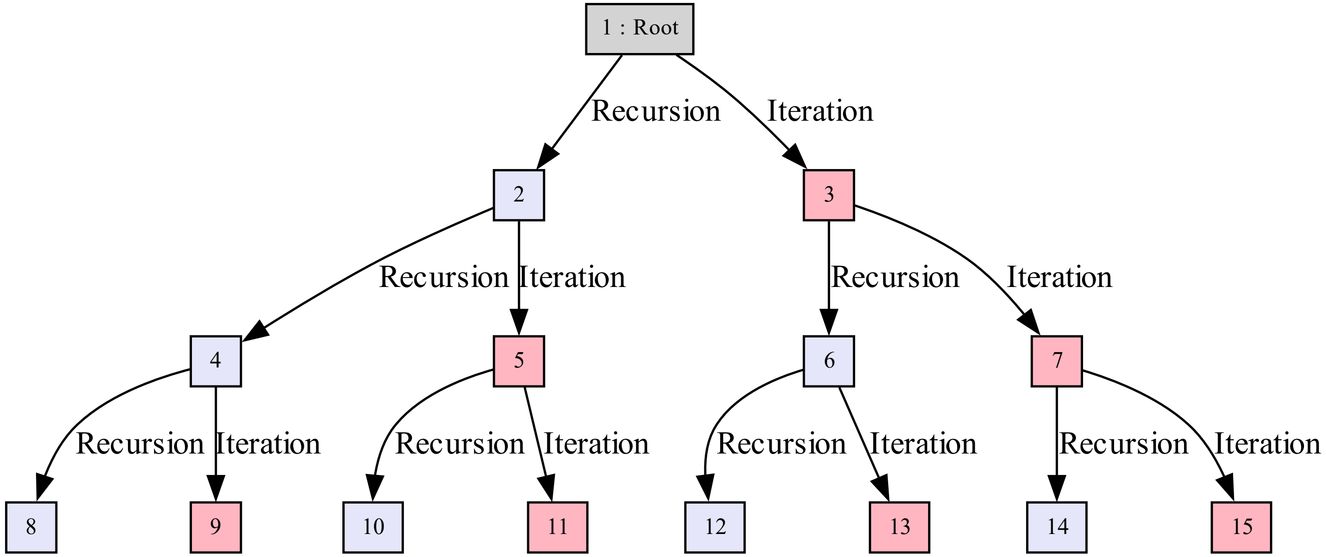
```asm
example::draw_and_report_area_dynamic:
push    r14
push    rbx
push    rax

mov     r14, rsi                ; Save the address of the vtable
mov     rbx, rdi                ; Save the address of a
call    qword ptr [rsi + 32] ; a.draw() is called via the vtable
mov     rdi, rbx
mov     rax, r14
add     rsp, 8
pop     rbx
pop     r14
jmp     qword ptr [rax + 24]    ; 🪱 Tail call optimized a.area() jump
```

Recursion and tail call optimization

```
pub enum Tree<T> {
    Node(T, Box<Tree<T>>, Box<Tree<T>>),
    Leaf(T),
}
use Tree::{Leaf, Node};

pub fn sum(tree: &Tree<u64>) -> u64 {
    match tree {
        Leaf(n) => *n,
        Node(n, left, right) => *n + sum(left) + sum(right),
    }
}
```

A recursive tail call is converted into a loop






```rust
pub fn sum(tree: &Tree<u64>) -> u64 {
    match tree {
        Leaf(n) => *n,
        Node(n, left, right) => *n + sum(left) + sum(right),
    }
}
```

Closures

```
pub fn make_quadratic(a: f64, b: f64, c: f64) -> impl Fn(f64) -> f64 {
    move |x| a*x*x + b*x + c
}
```

Closure memory layout

00 | a	5.0
08 | b	4.0
16 | c	3.0

Assembly of **make_quadratic**

```
example::make_quadratic:
mov     rax, rdi                      ; rax = Address of the closure
movsd   qword ptr [rdi], xmm0         ; closure.a = a
movsd   qword ptr [rdi + 8], xmm1     ; closure.b = b
movsd   qword ptr [rdi + 16], xmm2    ; closure.c = c
ret                                   ; Return address of closure in rax
```

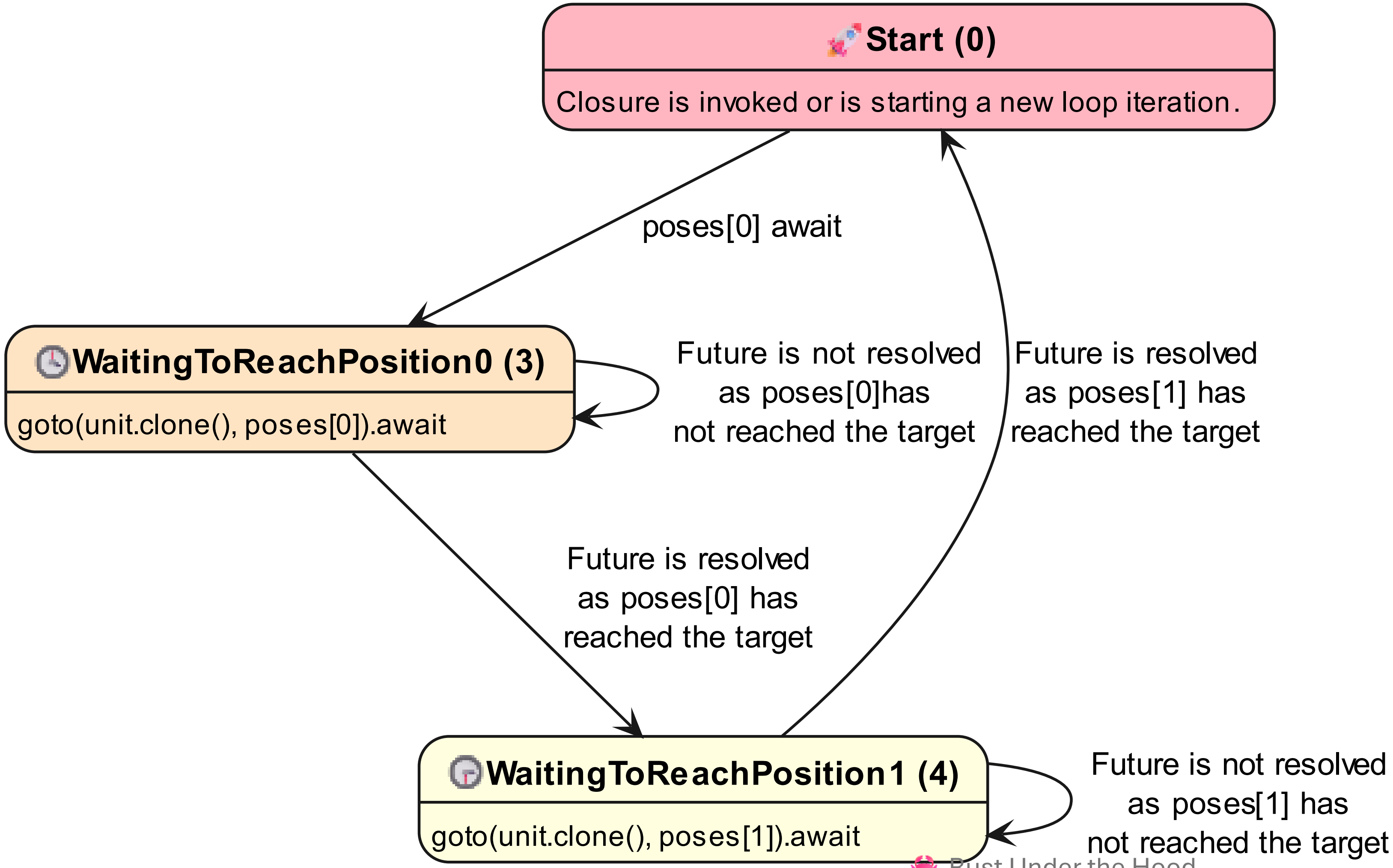
An infinite loop with async/await

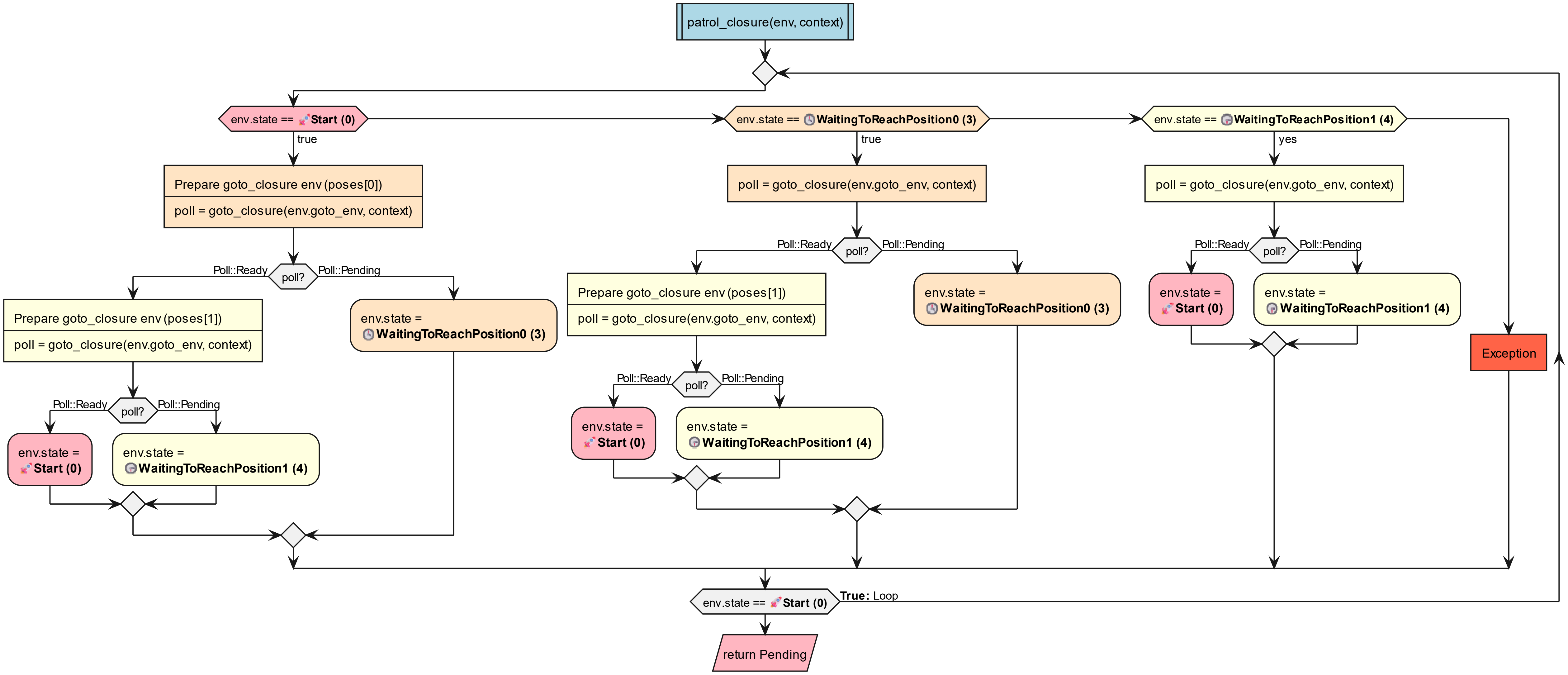
```rust
async fn patrol(unit: UnitRef, poses: [i32; 2]) {
    loop {
        goto(unit.clone(), poses[0]).await;
        goto(unit.clone(), poses[1]).await;
    }
}
```

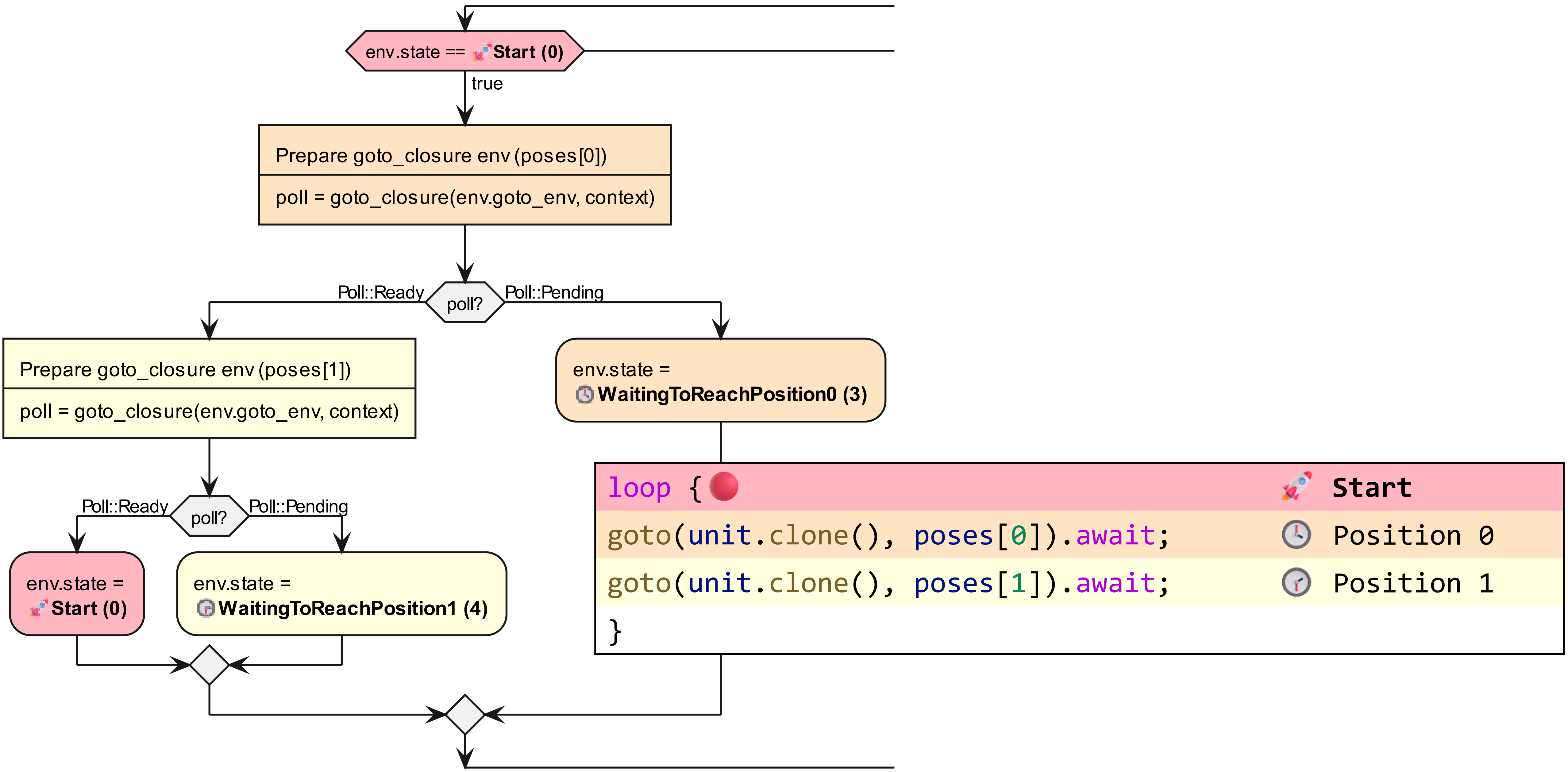
Credit: https://github.com/enlightware/simple-async-local-executor

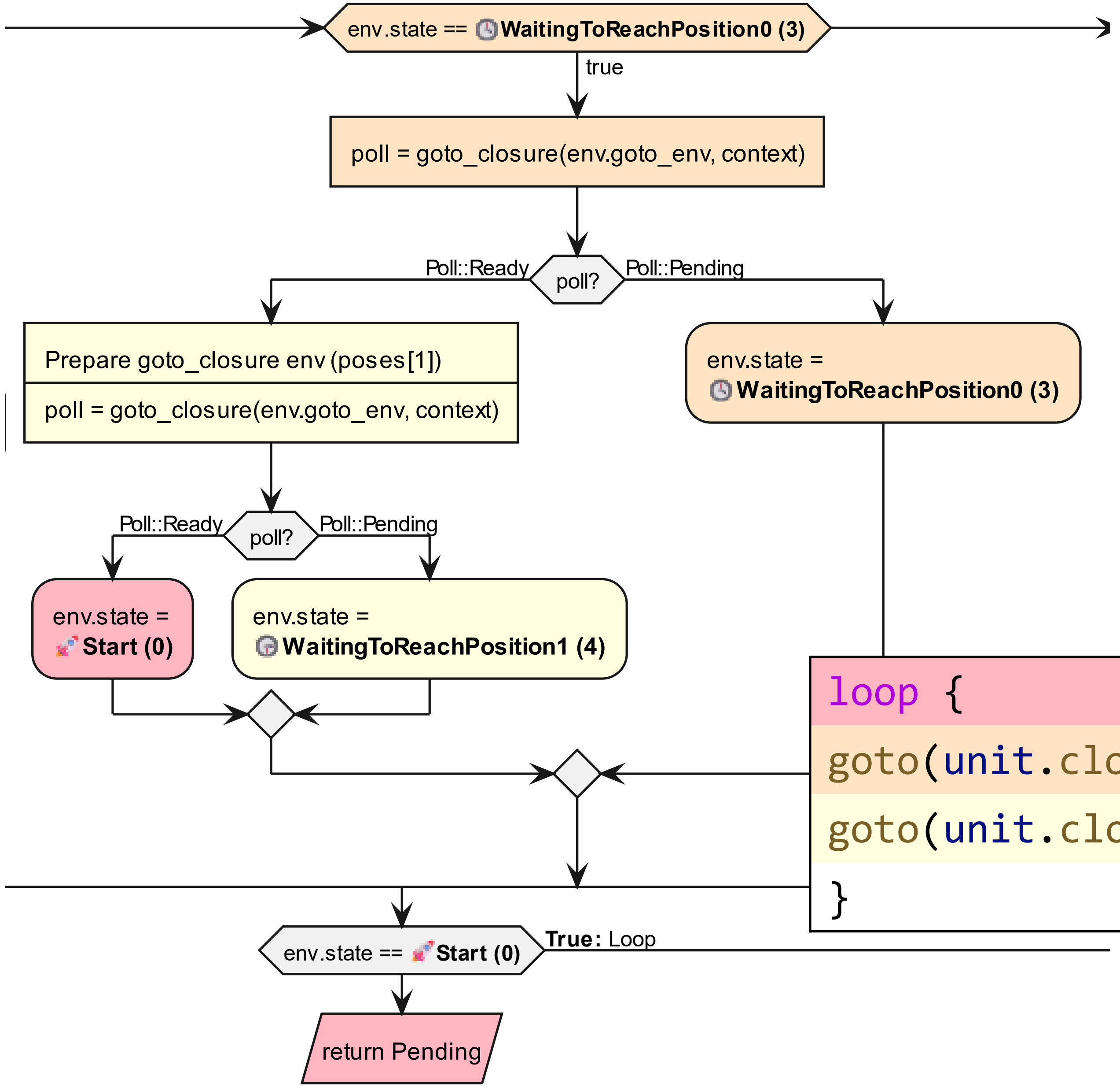
Async state machine

```rust
async fn patrol(unit: UnitRef, poses: [i32; 2]) {
    loop {
        goto(unit.clone(), poses[0]).await;
        goto(unit.clone(), poses[1]).await;
    }
}
```

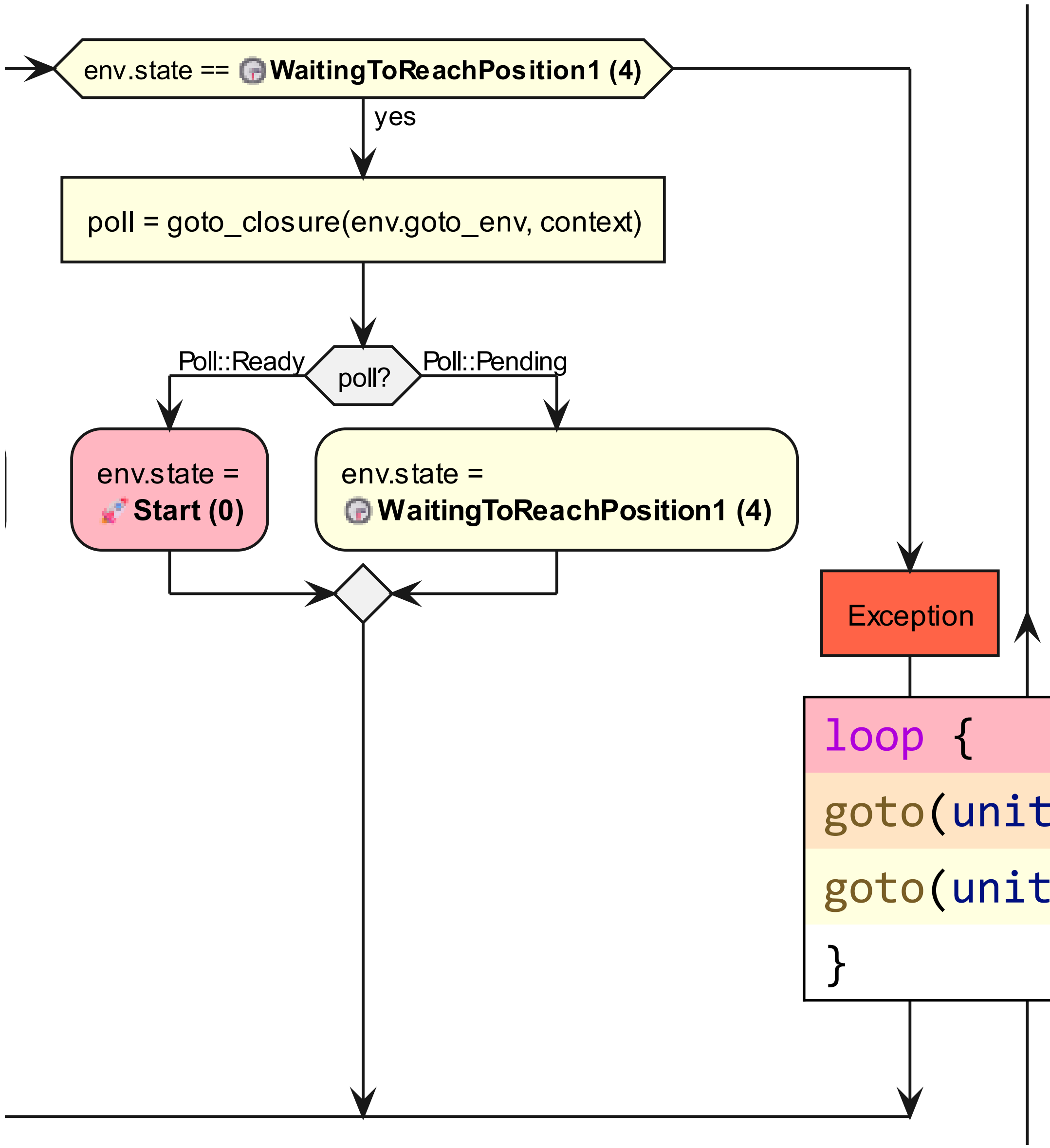



patrol_closure(env, context)
env.state == Start (0)
true
Prepare goto_closure env (poses[0])
poll = goto_closure(env.goto_env, context)
Poll::Ready
poll?
Poll::Pending
Prepare goto_closure env (poses[1])
poll = goto_closure(env.goto_env, context)
env.state = WaitingToReachPosition0 (3)
Poll::Ready
poll?
Poll::Pending
env.state = Start (0)
env.state = WaitingToReachPosition1 (4)
env.state == WaitingToReachPosition0 (3)
true
poll = goto_closure(env.goto_env, context)
Poll::Ready
poll?
Poll::Pending
Prepare goto_closure env (poses[1])
poll = goto_closure(env.goto_env, context)
env.state = WaitingToReachPosition0 (3)
Poll::Ready
poll?
Poll::Pending
env.state = Start (0)
env.state = WaitingToReachPosition1 (4)
env.state == WaitingToReachPosition1 (4)
yes
poll = goto_closure(env.goto_env, context)
Poll::Ready
poll?
Poll::Pending
env.state = Start (0)
env.state = WaitingToReachPosition1 (4)
Exception
env.state == Start (0)
True: Loop
return Pending


env.state == Start (0)
true
Prepare goto_closure env (poses[0])
poll = goto_closure(env.goto_env, context)
Poll::Ready
poll?
Poll::Pending
Prepare goto_closure env (poses[1])
poll = goto_closure(env.goto_env, context)
env.state =
WaitingToReachPosition0 (3)
Poll::Ready
poll?
Poll::Pending
env.state =
Start (0)
env.state =
WaitingToReachPosition1 (4)
loop {
Start
goto(unit.clone(), poses[0]).await;
Position 0
goto(unit.clone(), poses[1]).await;
Position 1
}

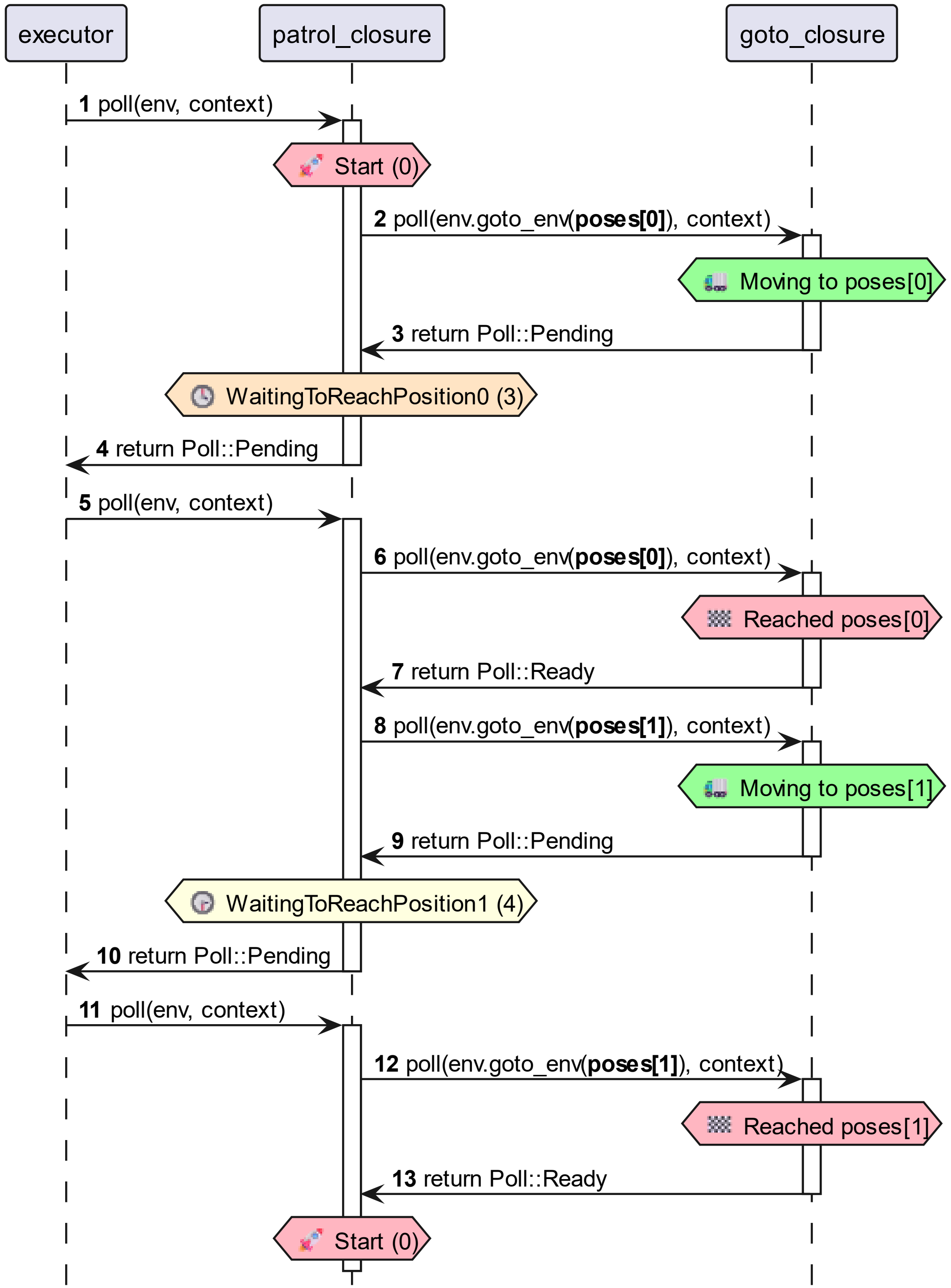



```
loop {                                         🚀 Start
goto(unit.clone(), poses[0]).await;   🔴       🕓 Position 0
goto(unit.clone(), poses[1]).await;            🕐 Position 1
}
```




```
loop {                                          🚀 Start
goto(unit.clone(), poses[0]).await;             Position 0
goto(unit.clone(), poses[1]).await;   🔴        Position 1
}
```

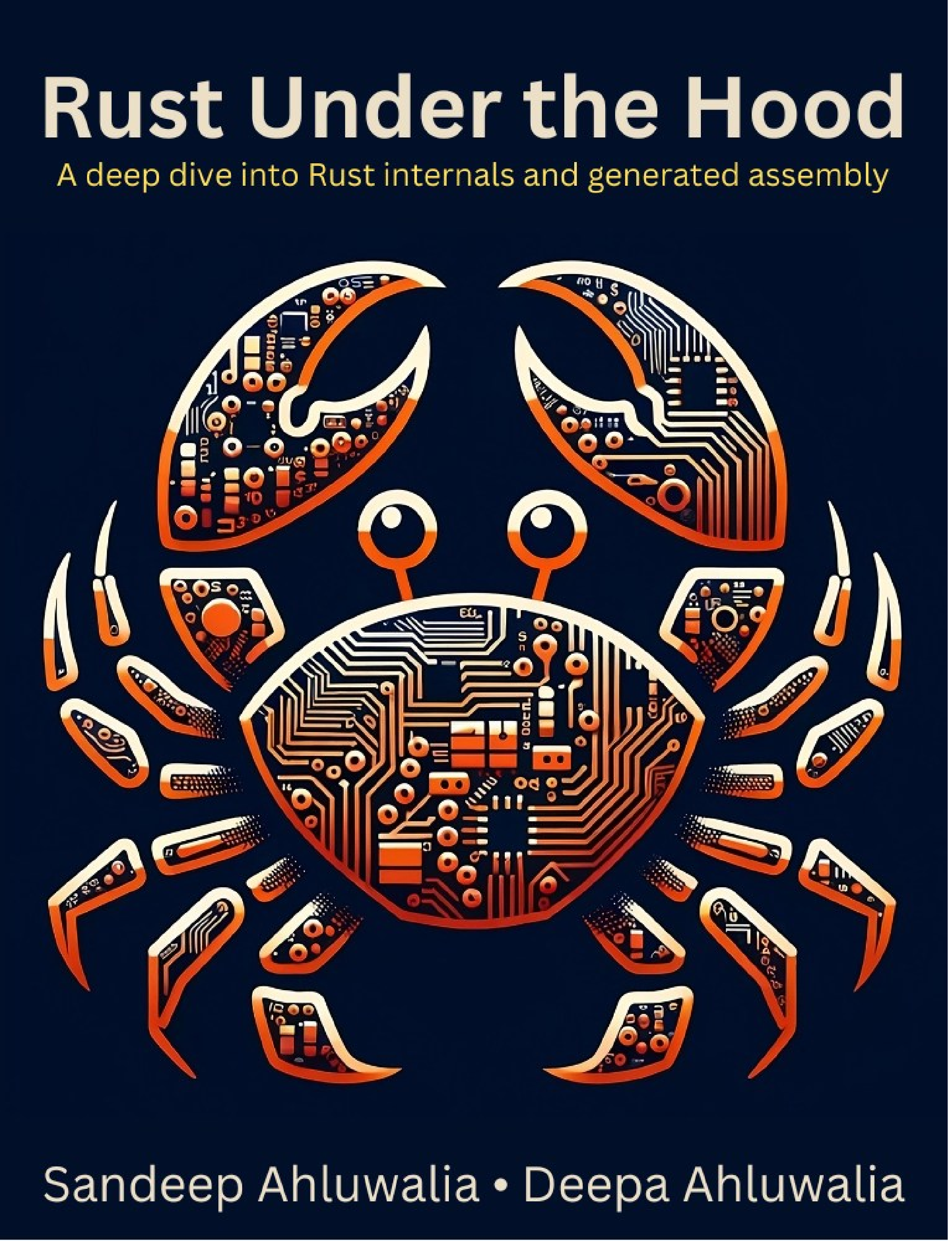
Async executor polling




```rust
async fn patrol(unit: UnitRef, poses: [i32; 2]) {
    loop {
        goto(unit.clone(), poses[0]).await;
        goto(unit.clone(), poses[1]).await;
    }
}
```

We covered

- Memory layout of enums, struct, vectors, strings, and arrays
- Pattern matching internals
- How smart pointers manage memory
- Tail call optimization and recursion
- Dynamic dispatch and vtables
- Functional programming is a zero-cost abstraction
- How closures capture the environment
- How async/await desugars into futures and state machines



Thank You

Rust Under the Hood is available at Amazon and PayHip

https://eventhelix.com/