Rust Language Updates: Arbitrary Self Types and In-place initialization

Xiangfei Ding

Kangrejos 2025, Oviedo, Spanien

Outline

- `#![feature(arbitrary_self_types)]` in 2025
  - The many, many and many ways to enable it
  - Trait evolution…?
  - The Great Split
- In-place initialization
  - What is it and why does it matter to us?
  - Also the many, many and many ways to enable it
  - Our roadmap to enable it

arbitrary_self_types

- Extension to types that a method receiver variable `self` can take
  - Such as `Box` and `Rc`

```
impl MyStruct {
    fn method1(&self) {}

    fn method2(self: &Self) {}

    fn method3(self: Box<Self>) {}

    fn method4(self: Pin<&mut MyStruct>) {}
}
```

`arbitrary_self_types` and why do we need it?

- Refined method receiver type encodes more constraints
  - `Pin<&mut T>, Arc<T>`
- Make Smart Pointers productive and useful
  - `#![feature(~~SmartPointer~~CoercePointee)]`
  - `CppRef<T>`

S. 25

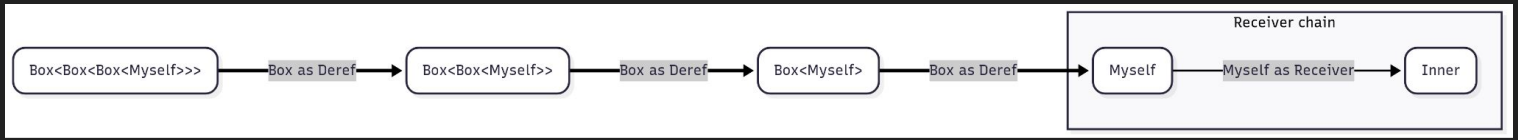
`arbitrary_self_types`: the method probing

- So when the code says `value.as_ref()`, how does it know which method is called?
- Suppose `value` is a smart pointer like `Arc` from kernel…
- Should we call method that is inherent to `Arc`? Or a trait method from AsRef because Arc implements it?
- Or alternatively we should dereference the Arc once and see if `as_ref` can be applied on the dereference data?

Arc<T>

`arbitrary_self_types`: the method probing (cont.)

- When only a method name is given, it is possible that an applicable method will appear as we repeatedly dereference the (smart) pointer along the way.
- `arbitrary_self_types` as of 2025: we probe deeper for applicable method, not by dereferencing, rather by chasing down the `Receiver` trait



`arbitrary_self_types`: the method probing (cont.)

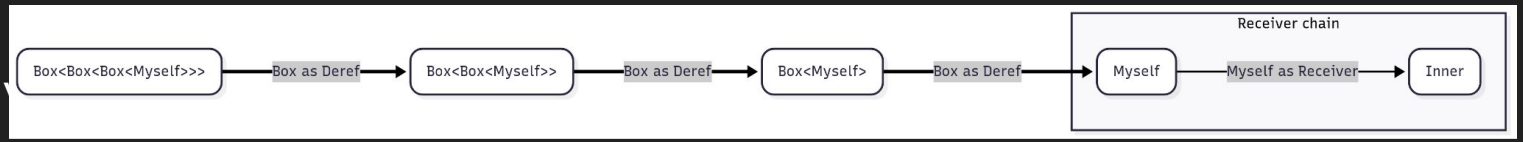
```
struct Inner;
impl Inner {
    fn resolve(self: Myself) {}
}
struct Myself;
impl Receiver for Myself {
    type Target = Inner;
}

let value: Box<Box<Box<Myself>>>;

value.resolve(); // <-- here
```

arbitrary_self_types: the method probing (cont.)

```
struct Inner;
impl Inner {
    fn resolve(self: Myself) {}
}
struct Myself;
impl Receiver for Myself {
    type Target = Inner;
}
let value = Myself;
value.resolve(); // <-- here
```



`arbitrary_self_types`: a historical design

- It turns out to be a design that has a path dependent on how this feature worked in the past
  - `Receiver` was not there
  - In its place is `Deref`
  - Types like `Box`, `Arc`, `Rc`, `Pin` are marked with a trait **`Legacy`**`Receiver`
- At the moment there is strong tie between Deref and Receiver
  - Conceptually if a type T implements Deref, T implements Receiver as well
  - Problem: Pin<T> is a valid method receiver under the **`Legacy`**`Receiver` mechanism, but it is not anymore under the `Receiver` mechanism
- And two traits are not fit to each other somehow…

`arbitrary_self_types`: many ways to do it

- April 2025: Make Receiver a supertrait of Deref?
  - `pub trait Deref: Receiver { .. }`
  - The idea is to just make it work for `Deref`
  - We need the next solver.
- June 2025: Okay, can we evolve the Deref trait to introduce this supertrait relationship through an language mechanism?
  - More on this later
- Benno, August 2025: two traits are not fit to each other somehow…

`arbitrary_self_types`: a better model

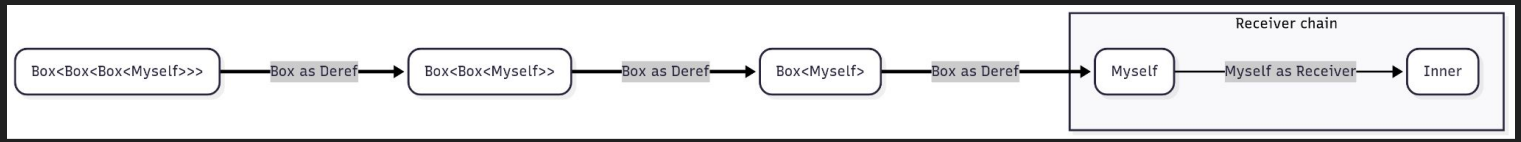
- Why does Receiver have to couple with Deref from the start?
  - `Deref`: a conforming type works like a pointer to another type
  - `Receiver`: a conforming type allows itself to
    - Act like a method receiver, being the type of the variable `self` and
    - Indirectly dictates the type of `Self`
- “Interesting” consequences
  - A method using `MutexGuard` as `self`?

```
fn method(self: MutexGuard<'_, Self>)
```

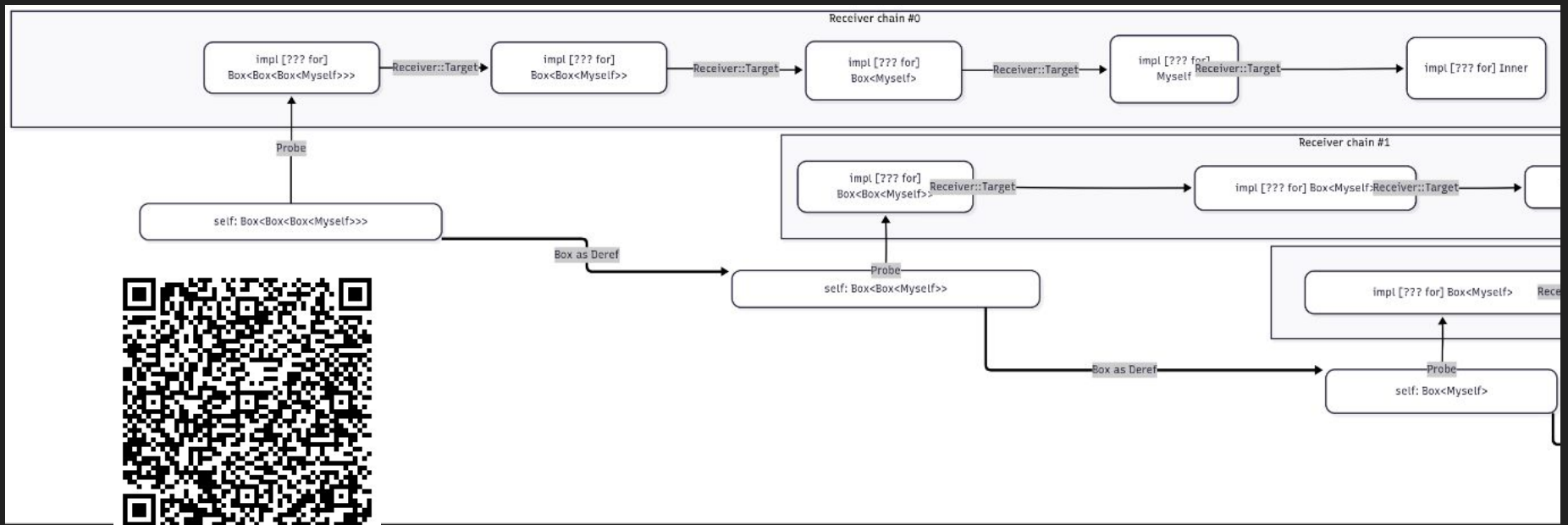
  - Deref target must be the same as Receiver target, but this rule can be too strict
- Let us separate them!

`arbitrary_self_types`: a better model

- Let us separate them!



`arbitrary_self_types`: a better model



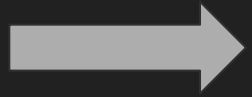
`arbitrary_self_types`: next step

- Fix diagnostics in #146095
- Provide a summary of changes to the method probing
- Document the language changes in the Rust Reference

Digression: trait evolution

- Trait evolution would help with landing arbitrary_self_types if we stay with the design to couple Deref and Receiver, but it is also useful so that impact on downstream trait implementors from a trait refactor is minimised.

```
trait MegaTrait {
    type SomeType;
    fn do_one_thing();
    fn do_another_thing();
}
```



RFC 3851 will allow us to refactor like this

```
trait SmallerTrait {
    type SomeType;
    fn do_one_thing();
}

trait MegaTrait: SmallerTrait
{
    auto impl SmallerTrait;
    fn do_another_thing();
}
```

In-place initialization

- Initializing data in a pre-allocated place of memory
  - Kernel: self-referential data structures like linked lists
  - `KBox::pin_init`
- Also important in every ecosystem when large data structures are concerned

In-place initialization

- We love `pin_init!` and we would like to make a language feature out of it
- A few basic design axioms for assessment
  - Explicit syntactical signal
  - Composability
  - Fallibility
  - Pinned place
  - Language-assisted value lifecycle management
    - Guaranteed clean-up on failure

In-place initialization: the many, many ways to enable it

- Three approaches
- The `init` expression (Alice, Benno)
  - Write your **`struct`** and **`enum`** expressions as usual
  - … and we compile it into an **`impl`** `PinInit` for you
  - Clearly defined trait interface

```
KBox::pin_init(init MyDriver {
    index: init_my_index(),
    data: [0u8; 2048] // big, but also initialized in-place
})
```

In-place initialization: the many many ways (cont.)

- out-pointer (Taylor Cramer. et al.)
  - Borrow checker has tracking on initialization state of `struct` fields

```
let _ = x.a; // Drop
x.a = MyStruct;
```

  - What is stopping us is a way to lend **out** uninitialized memory to write data into

```
let outptr = &out place;
outptr.a = MyStruct;
// place is initialized
```

- Guaranteed Named Emplacement, aka GNE (Olivier)
  - Still in conception phase
  - In code generation, function return places are implicitly passed down

In-place initialization: the many many ways matrix

	Composability	Fallibility	Explicit syntax	Pinnedness	Lifecycle mgmt
`init`	✓	?	✓	?	✓
outptr	✓	✓	✓	✓	?
GNE	?	✓	x		✓

In-place initialization: roadmap

- 2025: Continue with nightly experiment on `init` expression
- 2025-2026: Investigation into out-pointer
  - Formulate the proposal draft here at Kangrejos :construction:
- Follow-up with GNE proposals
- Implementation (?)

Q & A

Rust Language Updates: Arbitrary Self Types and In-place initialization

Xiangfei Ding

Kangrejos 2025, Oviedo, Spanien