Rust language evolutions

for better kernel developer experience

Xiangfei Ding (@wieDasDing)

Linux Plumbers Conference

11.12.2025, Tokyo

What shall we cover?

- Arbitrary self types mini-update
- In-place initialisation

Arbitrary self types

What is `self` type?

The *Context*

```
int method(struct context_t *ctxt, uint8_t *input) { .. }

method(&ctxt, &input);
```

```
fn method(ctxt: &mut Context, input: &u8) { .. }

method(&mut ctxt, &input);
```

```
impl Context {
    fn method(&mut self, input: &u8) {
        // self: &mut Context
    }
}
let mut ctxt: Context;
ctxt.method(&input);
```

Message-passing style

	Agent/Receiver	Message
C	`ctxt`	`method`
Rust	`self`	`method`

Issue: A reference counting receiver?

C

```c
// Invasive reference counting
void Context_refcount_bump(struct context_t *);
int method(struct context_t *ctxt, uint8_t *input) {
    Context_refcount_bump(ctxt);
}
```

Rust ... since 2017

```
impl Context {
    fn method(self: Arc<Context>, input: &u8) {
        //    ^~~~~~~~~~~~~~~~~~~
        let context_ref = Arc::clone(&self);
    }
}
```

Issue: Hello `Pin` my old friend.

```
impl Context {
    fn method(self: Pin<&mut Context>, input: &u8) {
        // Nice, you can't hold `Context` wrong now.
    }
}
```

Issue: Smart Pointers

```
#[derive(CoercePointee)]
struct Ptr<T: ?Sized>(..);

impl Context {
    fn method(self: Ptr<Context>, input: &u8) {
        //    ^~~~~~~~~~~~~~~~~~
    }
}
```

`Receiver` Trait

```
pub trait Receiver {
    type Target: ?Sized;
}
```

Message-passing style, with Rust characteristics

	Receiver	???	Message
C	`ctxt`	*not applicable*	`method`
Rust	`self`	`Context`	`method`
Rust	`self: Ptr<Context>`	`Context`	`method`

Issue: `Deref` / `DerefMut` trait

Generalised Indirection and Dereference

```
let x: &&Context = ..;
// This call works because
x.method();
// .. is lowered into ..
(*x).method();

let x: Box<Context> = ..;
// This call also works because
x.method();
// .. is lowered into ..
(*x).method();
// .. and further into ..
x.deref().method();
```

```rust
pub trait Deref {
    /// The type that `Self` can be dereferenced into.
    type Target: ?Sized;

    // This prescribes how to to
    fn deref(&self) -> &Self::Target;
}
```

```rust
impl<Ptr: Deref> Deref for Pin<Ptr>
where
    Ptr: Deref,
{
    type Target = <Ptr as Deref>::Target;
    fn deref(&self) -> &<Ptr as Deref>::Target { .. }
}

impl<P: ?Sized, T: ?Sized> Receiver for P
where
    P: Deref<Target = T>,
{
    type Target = T;
}
```

```
use kernel::types::Opaque;

impl Context {
//   ^~~~~~~
    fn method(self: Pin<&Opaque<Context>>, input: &u8) {
        //          ^~~~~~~~~~~~~~~~~~~~~
        //          ... but this dereferences to Opaque<Context>
        //              not Context
    }
}
```

The bane of strong `Deref`/`Receiver` coupling

Question 1: Do we always want `T: Receiver` if `T: Deref`?

Suspicious receivers like

- `std::vec::PeekMut`
- `DropGuard`
- `LazyLock`
- … various lock guards

Question 2: Is `Receiver::Target == Deref::Target` desirable?

There are missed opportunities.

A possibly better Rust message-passing style

- Access Handle: contracts, access, capability
- Context: the new **Agent** of the traditional message-passing
- Message: the program section that contracts are upheld and access is granted

Access Handle	Context	Dereferenced to	Access control, capability
`self`	`Context`	N/A	Owner
`&self` / `&mut self`	`Context`	`Context`	Borrow with access control
`self: Pin<&mut Context>`	`Context`	N/A	Pinned guarantee
`self: Pin<&Opaque<Context>>`	`Context`	N/A	Pinned guarantee and interoperability
`self: Arc<Context>`	`Context`	`method`	Reference counting
`self: BusDevice<Bus, Device>`	`Device`	`BusHandle<Bus>`	Bus protocol

Next step

- Provide updated method resolution in Rust Reference.
- Optimise the method resolution.
- An unstable feature gate to materialise the Rust message-passing style
  - `Receiver::Target == Deref::Target`
  - `Receiver::Target != Deref::Target`

In-place initialisation

What are we cooking?

- C++-inspired Guaranteed (Return) Value Emplacement, implicit emplacement
- `#[placing]` function, explicit emplacement
- `pin-init` but in standard library
- ...

Disclaimer

The following content does not constitute any emdorsement or render neither promise of design or implementation, nor preference to design or implementation.

It should Just Work™

Stick to minimal units of constructions

Minimal `unsafe`

Let's see how we can work out a good surface language feature

C

```c
struct Data {
    uint8_t a;
    uint8_t *b;
    // Call clean-ups when you are done with `Data`
    struct Droppy c;
} data;

data.a = 0;
data.b = &data.a;
data.c = (struct Droppy) {};
// Done!
```

Rust

```
struct Data {
    a: u8,
    b: *mut u8,
    c: Droppy,
}

let data: Data; // ERROR: `data` is uninitialised
data.a = 0; // ERROR: `data` is not initialised
data.b = &data.a; // ERROR
data.c = Droppy;
```

Error message can probably be better

Rust

```
let data: Data;
evil(&data);
//~^ ERROR `data.a` `data.b` and `data.c` are uninitialised
```

Let us relax initialisation checks

Rust

```rust
struct Data {
    a: u8,
    b: *mut u8,
    c: Droppy,
}

let data: Data;
// data is uninitialised
data.a = 0;
// data.a is initialised
data.b = &raw mut data.a;
// data.b is initialised
data.c = Droppy;
// data is initialised
let _ = &data;
```

Hand off initialisation to other subroutines

C

```c
struct Data {
    uint8_t a;
    uint8_t *b;
};

void init_for_me(struct Data *data) {
    data.a = 0;
    data.b = &data.a;
}
int try_init_for_me(struct Data *data) {
    data.a = 0;
    data.b = &data.a;
    return 0;
}
```

Rust

```rust
struct Data { .. }

let data: Data;
fn init_for_me(data: & ?) {
    todo!()
}
fn try_init_for_me(data: & ?) -> Result<?, ?> {
    todo!()
}

init_for_me(todo!());
try_init_for_me(todo!())?;
// data is initialised, but how exactly?
```

Let us cook some new types

- A pointer to a possibly uninitialised place
  - `&uninit Data`
  - ... should work like an `&mut Data` but it is known to be uninitialised to borrow checker
  - valid to write into, but invalid to read from until the first initialisation

It is not perfect. We still have some quirks.

Rust

```rust
fn init_for_me<'a>(data: &'a uninit Data) {
    data.a = 0;
    data.b = &raw mut data.a;
    data.c = Droppy;
    // Magic?
}

let data: Data<'_>;
init_for_me(&uninit data);
// Hmm, okay-ish?
try_init_for_me(todo!())?; // ???
```

Rust

```
// Expand try_init_for_me(&uninit data)?; into ...
match try_init_for_me(&uninit data) {
    Ok(()) => {
        // data is initialised in this arm
        // Magic?
    }
    Err(err) => {
        // data is *uninitialised* in this arm
        // Even more magic?
    }
}
```

Sometimes you just have to hold onto uninitialised data longer before you complete further computation.

C

```c
while (1) {
    error = BIO_read(io_handle, &buffer.data, buffer.size);
    if (error) {
        error = CTX_get_real_error(io_handle);
        if (error == EAGAIN || error == EWOULDBLOCK) {
            continue;
        }
        if (error == CTX_RENEGOTIATE) {
            CTX_reset(io_handle);
            continue;
        }
        // true error
        return error;
    }
}
```

Let us cook some new types (cont.)

- A pointer to a definitely initialised place
  - `&own Data`
  - ... should work like an `&mut Data`;
  - ... can be derived from a `&uninit Data` whose pointee is known to be initialised;
  - ... calls destructor when dropped, like an owned value of `Data`.
- And a mechanism to mark a `&uninit Data` initialised with a `&own Data`

```
let uninited: &uninit Data;
let inited: &own Data;

*uninited <- inited;
```

Rust

```rust
fn try_init_a(a: &uninit u8)
    -> Result<&own u8, ()>
{
    *a = 0;
    Ok(a)
}
fn try_init_b<'b>(a: &u8, b: &'b uninit *mut u8)
    -> Result<&'b own *mut u8, ()>
{
    *b = &raw mut *a;
    Ok(b)
}
```

Rust

```
let data: Data;
data.a <- try_init_a(&uninit data.a)?;
data.b <- try_init_b(&data.a, &uninit data.b)?;
if data.a > 0 {
    // initialisation abandoned

    return;
    // Note: no destructor call on data.c because
    // it is known to be uninitialised here
} else {
    data.c = Droppy;
}
// data is initialised
```

Emplacing statements with `<-`

```
let data: Data;
data <- Data {
    a: 0,
    b: &raw mut a,
};
data.c = Droppy;
// .. or ..
data <- Data {
    a: 0,
    b: &raw mut a,
    c <- Droppy,
};

let data: [u8; 65536];
data <- [0; 65536]; // No duplicated allocation of 65536 bytes
```

DRY: `PinInit` trait is a good interface

```
/// Sketch of `PinInit`
trait PinInit<T> {
    unsafe fn pin_init(self, slot: Pin<&uninit T>) -> Pin<&own T>;
}

let data: MyData;
// so instead of ..
data.a <- make_a(&uninit data.a);
// .. we just go with ..
let pin_init_data: impl PinInit<A> = make_pin_init_out_of(make_a);
data.a <- pin_init_data;
// .. or ..
data.a <- make_a(@);
// make_a(@) is an `impl PinInit` generated by use of the sigil @
```

- C++-inspired Guaranteed (Return) Value Emplacement, implicit emplacement
  - Still very much sought after
- Be the foundation of
  - `#[placing]` function, explicit emplacement
  - `pin-init` in standard library

WDYT?

Rust language evolutions

for better kernel developer experience

Xiangfei Ding (@wieDasDing)

Happy Holidays and see you next time