pin-init self references

+ advanced lifetime patterns in practice

Gary Guo

Kangrejos 2026

16/09/2026, Las Palmas

Covariance

We say `Type<T>` is covariant over `T`, if `Type<Sub>` is a subtype of `Type<Super>`.

Example: `&'a T` is covariant over both `'a` and `T`.

- A longer living lifetime is a subtype of a shorter living one (hence, `'static` subtypes all lifetimes)
- This enables you to use `&'static T` when code expects `&'a T`.

Variance when **producing** value.

Background knowledge
— Covariance

In Rust, subtyping relation only exists for types that differ by lifetime only.

The fact that reference is covariant over its lifetime is what allows you to shorten the lifetime of reference.

It is covariant over `T` because `T` can itself contains a lifetime; so if you have nested references, you can shorten nested ones, too.

Contravariance

We say `Type<T>` is contravariant over `T`, if `Type<Super>` is a subtype of `Type<Sub>`.

Example: function argument: `fn(T) -> U` is contravariant over `T`

- So `fn(&'a str)` subtypes `fn(&'static str)`.

Variance when **accepting** value.

Background knowledge
— Contravariance

This makes sense because that it's identical to performing a covariant subtyping on the caller side.

Invariance

We say `Type<T>` is invariant over `T` if so `Type<Sub>` and `Type<Super>` does not subtype (unless `Sub = Super`).

Example: `&'a mut T` is invariant over `T` (note it is still covariant over `'a`); `Mutex<T>` is invariant over `T` as well.

It is important to prevent this code from compiling

```
let mut long: &'static u32 = ...;
let short: &'shorter u32 = ...;
*(&mut long as &mut &'shorter u32) = short;
```

Another source of invariance is trait and generic associated types (GATs).

Background knowledge
— Invariance

One way to conceptualize this is to think a mutable container can both produce and accept value. The reading part is covariant, while the writing part is contravariant, so the combined result is invariant.

For completeness, there is also bivariant, when a type is both covariant and contravariant.

Rust generally forbids presence of bivariant parameters in types. When you see “type parameter is unused” error message and needs to add `PhantomData`, it’s Rust inferring bivariant and thus complaining.

Wellformedness

In Rust, types need to be wellformed. It means that all bounds on the type definition must be satisfied:

```
struct Foo<T: Clone>(T);

type X = Foo<Bar>; // Require `Bar: Clone` to be wellformed.
```

Background knowledge
— Wellformedness

This may looks trivial, but WF bounds are different, say, impl bounds. WF bounds are required even to *mention* a type at all.

Wellformedness

For lifetimes, an important wellformedness rule is that `&'a T` requires `T: 'a` to be wellformed. So

```
struct Foo<'a, T>(&'a T);
```

requires `T: 'a` to be wellformed.

This is needed because `T` itself is not necessarily an owned type and can contain lifetime too.

You rarely need to specify this yourself, because Rust add these as implied bounds whenever such types are mentioned in the signature.

One case where this does need explicit notation is for `'static` lifetime. In many cases, when Rust tells you that a type parameter needs to be `'static` but you cannot find where the bound is required explicitly, it's the wellformedness in play.

Wellformedness

For lifetimes, an important wellformedness rule is that `&'a T` requires `T: 'a` to be wellformed. So

```
struct Foo<'a, T>(&'a T);
```

requires `T: 'a` to be wellformed.

However, if you write code like this:

```
fn foo<'a, T>() {
    let _: &'a T = todo!();
}
```

then compiler will yell because it lacks the implied bound necessary for WFness.

Types appearing in function signature have their WF bounds *implied*, where types appearing in the body have their WF bounds *checked*.

Higher-rank polymorphism

Before pin-init self-reference, let's first take a look about higher-rank polymorphism.

Rank-1 polymorphism

```
fn foo<'a: 'a>(x: &'a u32) {}
```

is of type `fn(&'a u32)` for all `'a`. However it has no higher-rank polymorphism, so

```
fn test() {
    let bar = foo;
    bar(&identity(1));
    bar(&identity(1));
}
```

does not work.

With only rank-1 polymorphism, while a function can be polymorphic over all lifetimes, we cannot talk about the fact of “a function that is polymorphic over lifetime” in the type system itself.

A `'a: 'b` bound makes the lifetime early-bound, which is a way to opt-out from higher-rank polymorphism in Rust. When `bar = foo` assignment occurs, `bar` needs a type, so it `bar` needs to pick a single `'a`.

Higher-rank polymorphism

```
fn foo<'a>(x: &'a u32) {}
```

is of type `for<'a> fn(&'a u32)`. Which manifest itself in the type system, so

```
fn test() {
    let bar = foo;
    bar(&identity(1));
    bar(&identity(1));
}
```

works.

Good thing is Rust does have higher-rank polymorphism.

One restriction is that the universal quantification applies on lifetimes only.

Higher-rank trait bounds

This polymorphism is named “higher-rank trait bounds” (HRTB):

```
fn foo<T>() where T: for<'a> SomeTrait<'a> {}
```

Despite the name, it can also appear in types too. Function pointer types:

```
type X = for<'a> fn(&'a u32);
```

and dynamic types:

```
type X = dyn for<'a> SomeTrait<'a>;
```


But what about other types? What if we want to write code that is itself generic over lifetimes?

Device resource lifetimes

Devices have resources, and the resources may only be accessed when the device is **bound**.

`DevRes` was our original solution

- Use `Arc` internally to share storage between driver-core and driver.
- Use `Revocable` to prevent continued access after unbind (return `Err`)

Higher-rank polymorphism
— Device resource lifetimes

Let's first begin with why we need lifetime in the kernel.

After device **unbound**, it may be unplugged, so access will be UAF. It may be rebound to another driver, so access will cause concurrency issue.

`Revocable` would introduce fallible access paths to all code. So we have a method, that if you provide a proof that the device is bound, you may access the resource without failure path.

This solution stays for quite a while until earlier this year a thread changed this [link] when we have a need for accessing resource within destructor.

Device resource lifetimes

Devices have resources, and the resources may only be accessed when the device is **bound**.

`DevRes` was our original solution

- Use `Arc` internally to share storage between driver-core and driver.
- Use `Revocable` to prevent continued access after unbind (return `Err`)

Alternative: Lifetime

Higher-rank polymorphism
— Device resource lifetimes

It was quite surprising in hindsight that we didn't go this approach to begin with. But naturally lifetime is the perfect solution.

There is a tendency to avoid lifetimes, but… it's mostly due to lack of infrastructure to deal with it well.

A need for higher-rank types

Auxiliary device has a pattern where it needs the ability to go from a type-erased `auxiliary::Device` to obtain data that the auxiliary device registration attaches to it.

```
impl auxiliary::Device {
  fn registration_data<'this, T: Any>(&'this self) -> &'this T;
}
```


Auxiliary bus is a bus where a driver exposes an interface to another. When the secondary driver binds, it receives a type-erased `auxiliary::Device`, so if it needs to access the API, it needs to undo the type erasure.

Without a lifetime, we use `T: Any` and perform downcasts when the access is needed.

A need for higher-rank types

Auxiliary device has a pattern where it needs the ability to go from a type-erased `auxiliary::Device` to obtain data that the auxiliary device registration attaches to it.

```
impl auxiliary::Device {
  fn registration_data<'this, T: Any>(&'this self) -> &'this T;
}
```

With lifetime, we need:

```
impl auxiliary::Device {
  fn registration_data<'this, for<'a> T<'a>>(&'this self) -> &T<'this>;
}
```

and this needs higher-rank types.

Auxiliary bus is a bus where a driver exposes an interface to another. When the secondary driver binds, it receives a type-erased `auxiliary::Device`, so if it needs to access the API, it needs to undo the type erasure.

Without a lifetime, we use `T: Any` and perform downcasts when the access is needed.

This particular example assumes a covariant `'a`.

Polyfill with HRTB

```rust
trait WithLt<'a> {
    type Of;
}

fn registration_data<'this, F>(&'this self) -> &'this <F as WithLt<'this>>::Of
where
  F: for<'a> WithLt<'a>;
```


The bound requires impl for *all* lifetimes, and the function choose a specific one.

Polyfill with HRTB

```
trait ForLt {
    type Of<'a>;
}

impl<T: for<'a> WithLt<'a>> ForLt for T {
    type Of<'a> = <T as WithLt<'a>>::Of;
}

fn registration_data<'this, F: ForLt>(&'this self) -> &'this F::Of<'this>;
```


Once you have the base `WithLt` impl, you can use GAT to make it nicer.

Manually implementing `ForLt` for each type you want to talk about is tedious. Can we somehow implement such traits automatically without defining additional types?

Polyfill with HRTB

Functions are higher-rank, so surely we can?

```
impl<'a, T, F> WithLt<'a> for F
where
    F: FnOnce(&'a ()) -> T,
{
    type Of = T;
}
```

Now `for<'a> fn(&'a ()) -> Foo<'a>` implements `ForLt`, where `ForLt::Of<'a>` is `Foo<'a>`!

It works for most of the cases, including the previous example. But in callback based APIs, it falls apart weirdly.

Polyfill with HRTB

Functions are higher-rank, so surely we can?

```
impl<'a, T, F> WithLt<'a> for F
where
    F: FnOnce(&'a ()) -> T,
{
    type Of = T;
}

fn my_api<F: ForLt>(_f: impl for<'a> FnOnce(&'a ()) -> F::Of<'a>) {}

fn main() {
    my_api::<fn(&()) -> &()>(|s| s);
}
```


It works for most of the cases, including the previous example. But in callback based APIs, it falls apart weirdly.

Implementing `ForLt`

```
error: lifetime may not live long enough
  --> for_lt_fn.rs:27:34
   |
27 |     my_api::<fn(&()) -> &()>(|s| s);
   |                              -- ^ returning this value requires that `'1` must outlive `'2`
   |                              ||
   |                              |return type of closure is &'2 ()
   |                              has type `&'1 ()`

error[E0308]: mismatched types
  --> for_lt_fn.rs:27:5
   |
27 |     my_api::<fn(&()) -> &()>(|s| s);
   |     ^^^^^^^^^^^^^^^^^^^^^^^^ one type is more general than the other
   |
   = note: expected reference `&()`
              found reference `&'a ()`
```


Fixed in the next solver, but until then?

Implementing `ForLt`

Daniel Henry-Mantilla has an alternative trick in his `higher-kinded-types` crate.

```
dyn for<'a> WithLt<'a, Of = &'a ()>
```

Which also produce a `ForLt` impl, but for some reason does not have the lifetime issue!

Kernel's `ForLt!()` macro utilize this trick, so you can write

```
ForLt!(NovaCoreApi<'_>)
```

in functions that want polymorphism over lifetime-generic type.

Kernel also provides a `CovariantForLt!()` macro in case you want to establish an additional property of covariance.

Unsound corners of the language

The compiler often confuses between whether the implied bound from WFness is to be assumed or checked.

The following is accepted by the compiler:

```
dyn for<'a> WithLt<'a, Of = &'static &'a ()>
```

and this:

```
type X = for<'a> fn(&'a ()) -> &'static &'a ();
```

You cannot produce a value of these type (witness). But the bare existence of that type allow the compiler to assume *any* `'a` can outlive `'static`.

We are unfortunately hitting a quite unsound corner of language.

The same applies to the fn encoding too.

We have some additional WF checks in the macro to workaround this to prevent you from writing this.

Creating self references safely

If the talk is about pin-init self-reference, why do I talk in such detail about `ForLt!`? It turns out they are closely related!

Device resource lifetimes, take 2

Previous:

```
struct MyDriver {
  dev: ARef<Device>,
  bar: Arc<Devres<Bar<BAR_SIZE>>>,
  other_stuff: SomethingThatNeedsBar,
}
```

After:

```
struct MyDriver<'a> {
  dev: &'a Device,
  bar: Arc<Bar<'a, BAR_SIZE>>,
  other_stuff: SomethingThatNeedsBar,
}
```


With device lifetime alone, the issue is partly solved.

In most cases, when a driver asks for a resource, it doesn't ask for it to just be destroyed at cleanup. It needs to use it!

Because we need to feed the resources to components that need to use it, we end up having to reference count every single resource, turning something of static scope into a runtime destruction paths.

Device resource lifetimes, take 2

```
#[pin_data]
struct NovaCore<'bound> {
  reg: auxiliary::Registration<'bound, ForLt!(NovaCoreApi<'_>)>,
  gpu: Gpu<'bound>,
  bar: Bar<'bound, BAR0_SIZE>,
}

fn probe<'bound>(pdev: &'bound Device<Core<'_>>) ->
    impl PinInit<NovaCore<'bound>, Error> + use<'bound>
{
    try_pin_init!(NovaCore {
        bar: pdev.iomap_region_sized::<BAR0_SIZE>(0, c"nova-core/bar0")?,
        // SAFETY: cheating to not having to do reference counting...
        gpu <- Gpu::new(pdev, unsafe { &*core::ptr::from_ref(bar) }),
        reg: {
            let gpu = unsafe { Pin::new_unchecked(&*core::ptr::from_ref(gpu.as_ref().get_ref())) };
            auxiliary::Registration::new_with_lt(
            pdev.as_ref(), c"nova-drm", 0, c"nova-core",
            pin_init!(NovaCoreApi { gpu, pdev }),
        )?,
    })
}
```


Without a good solution, you end up with either a lot of unsafes (as in Nova), or a lot of reference counting (as in Tyr).

You can see, the code already uses `pin-init`.

If you consider what pin-init does, it is really just about self-references. So far, we focus on C FFI so you always create self-references unsafely, so we can construct things like `Mutex` without using unsafe. Naturally, the next step is safe creation of them.

Prior art

Some existing crates that attempted at this:

- moveit
- rental
- yoke
- ouroboros

Self-reference is nothing novel. What is different this time?

Kernel's constraints

- Ergonomic
- No allocation
- Soundness

- Ergonomic
  - The result should look like normal Rust
- No allocation
  - This rules out almost all existing solutions
  - You simply cannot allocate in many context.
  - Implicit allocation is unacceptable.
  - Why allocate when you don't need to?
- Soundness
  - Many of self-referential crates have unsound corners
  - You also run into unsound corners of the Rust language too

Kernel's constraints

- Ergonomic
- No allocation
- Soundness ⟸ Most my brain cells are spent on this category

Creating self references safely
— Kernel's constraints

- Ergonomic
  - The result should look like normal Rust
- No allocation
  - This rules out almost all existing solutions
  - You simply cannot allocate in many context.
  - Implicit allocation is unacceptable.
  - Why allocate when you don't need to?
- Soundness
  - Many of self-referential crates have unsound corners
  - You also run into unsound corners of the Rust language too

Example

```
#[pin_data]
struct NovaCore<'bound> {
    reg: auxiliary::Registration<'gpu, ForLt!(NovaCoreApi<'_>)>,
    gpu: Gpu<'bar>,
    bar: Bar<'bound, BAR0_SIZE>,
}

fn probe<'bound>(pdev: &'bound Device<Core<'_>>) ->
    impl PinInit<NovaCore<'bound>, Error> + use<'bound>
{
    try_pin_init!(NovaCore {
        bar: pdev.iomap_region_sized::<BAR0_SIZE>(0, c"nova-core/bar0")?,
        gpu <- Gpu::new(pdev, bar),
        reg: auxiliary::Registration::new_with_lt(
            pdev.as_ref(), c"nova-drm", 0, c"nova-core",
            pin_init!(NovaCoreApi { gpu, pdev }),
        )?,
    })
}
```

Just like ordinary use of pin-init, **no extra syntax** terms and conditions apply.

Defining self-referential structs

Explicit notation inspired by `ouroboros`.

```
#[pin_data]
struct MyStruct {
    // By default, the borrowing is inferred by looking at all unbound lifetime in the type.
    inferred_borrow: &'owner String,

    // It is possible to explicitly specify the borrow with `#[borrows(field_name)]` syntax.
    #[borrows(owner)]
    explicit_borrow: &'owner String,

    // Shared borrow is the default, it is possible to borrow mutably with explicit annotation.
    #[borrows(mut mutable_owner)]
    mutable_borrow: &'mutable_owner String,

    owner: String,
    mutable_owner: String,
}
```


The explicit notation is a tentative design and is subject to change. It won't be included in the MVP.

Defining self-referential structs

```
#[pin_data]
struct MyStruct {
    // By default, covariance is implied (and checked)
    covariant: &'owner String,

    // You can annotate still.
    #[covariant]
    explicit_covariant: &'owner String,

    #[not_covariant]
    not_covariant: Mutex<&'owner String>,

    owner: String,
    mutable_owner: String,
}
```

About field lifetimes

The annotation syntax is heavily inspired by ouroboros, but I didn't use the `'this` syntax, because it cannot be soundly used if borrowed field is not covariant.

```
#[pin_data]
struct MySelfRefStruct<'a> {
    #[borrows(res_a, res_b)]
    // Cannot be `&'res_b ResourceB<'res_b>`
    res_b_ref: &'res_b ResourceB<'res_a>,
    #[borrows(res_a)]
    #[not_covariant]
    res_b: ResourceB<'res_a>,
    res_a: ResourceA<'a>,
}
```

Initialising a self-referential struct

Intentionally left blank, because you use it exactly like normal pin-init.

Initialising a self-referential struct

It will just work.

```
pin_init!(SelfRef {
    str: "hello world".to_owned(),
    part: &str[..5],
})
```

I consider this is as the biggest improvement over existing crates.

Intentionally left blank, because you use it exactly like normal pin-init.

Using the self-referential struct

For this part, it will be different from normal structs, because the self-referential fields don't really exist in the language:

```
let s: Pin<&mut SelfRef> = /* ... */;

// Shared accessors
let field = s.field_name(); // Covariant
s.with_field_name(|field| /* ... */); // Invariant

// Mutable accessor
s.with_project(|project| project.field_name = /* ... */); // Any variance
```

Probably as best as you can get without being a language feature?

Kernel's constraints

- Ergonomic ✅
- No allocation
- Soundness

How is this expanded?

Conceptually expanded like this (unsound, will explain later)

```
struct MyStruct {
    borrow: &'static String,
    owner: String,
}

struct MyStructPinDataLt<'owner>(..); // ZST helper for `pin_init!()` macro. Slot projection.

impl<'owner> MyStructPinDataLt<'owner> {
    // After initializing, gives out `&'owner String`.
    fn owner(self, slot: *mut MyStruct) -> SelfRefSlot<'owner, String>;
    fn borrow(self, slot: *mut MyStruct) -> Slot<&'owner String>;
}

fn make_init(
    f: impl for<'owner> FnOnce(slot: *mut MyStruct, data: MyStructPinDataLt<'owner>))
) -> impl PinInit<MyStruct>;
```

How is this expanded?

Accessor side is the common HRTB pattern that allows introducing new lifetime.

```
impl MyStruct {
    // Covariant subtyping.
    fn borrow<'this>(&'this self) -> &'this &'this String;

    fn with_borrow<'this, R>(&'this self,
        f: impl for<'owner> FnOnce(&'owner String) -> R
    ) -> R;

    fn with_project<'this, R>(self: Pin<&'this mut Self>,
        f: impl for<'owner> FnOnce(ProjectionLt<'this, 'owner>) -> R
    ) -> R;
}

struct ProjectionLt<'this, 'owner> {
    borrow: &'this mut &'owner String,
    // This gains it's dedicated field lifetime, so you can make assignment outside init!
    owner: &'owner String, // Downgraded to shared borrow, even in mutable projection.
}
```

Kernel's constraints

- Ergonomic ✅
- No allocation ✅
- Soundness

The journey to soundness

Struct expansion

```
struct MyStruct {
    borrow: &'static String,
    owner: String,
}
```

Struct expansion

```
struct MyStruct {
    borrow: &'static String,
    owner: String,
}
```

This is unsound because user can access `self.borrow` directly and get a static reference.

So, wrap it!

Block direct access

```rust
// Define in pin-init crate, no accessor available.
#[repr(transparent)]
struct Erase<T>(T);

#[pin_data]
struct MyStruct {
    borrow: Erase<&'static String>,
    owner: String,
}
```

Sound now?

Block direct access

```
// Define in pin-init crate, no accessor available.
#[repr(transparent)]
struct Erase<T>(T);

#[pin_data]
struct MyStruct {
    borrow: Erase<&'static String>,
    owner: String,
}
```

Sound now?

This is unsound because of auto traits can leak through and can observe the erased type!

```
struct LtSpec<'a>(...);
unsafe impl Send for LtSpec<'static> {}
```

Prevent lifetime specialization

To prevent lifetime specialization, we must be able to specify "for all lifetime" constraints in the type system $\Longrightarrow$ higher-rank polymorphism!

```
trait EraseLt {
    type Erased;
}
impl<T> EraseLt for (T,) {
    type Erased = T;
}
impl<T> EraseLt for T where
    T: for<'a> FnOnce<(&'a (),), Output: EraseLt>,
{
    type Erased = <<T as FnOnce<(&'static (),)>>::Output as EraseLt>::Erased;
}
```

Now, `for<'a> fn(&'a ()) -> for<'b> fn(&'b ()) -> (Foo<'a, 'b>,)` would be our way of encoding the information `for<'a, 'b> Foo<'a, 'b>`.

Prevent lifetime specialization

Now use HRTB to assert `for<'a, 'b> Foo<'a, 'b>: Send` (same for `Sync`)

```
#[repr(transparent)]
pub struct Erase<F: EraseLt>(F::Erased);

unsafe impl<T: Send> Send for Erase<(T,)> {}

// Peel one lifetime at a time...
unsafe impl<F: EraseLt> Send for Erase<F>
where
    F: for<'a> FnOnce<(&'a (),), Output: EraseLt>,
    for<'a> Erase<<F as FnOnce<(&'a (),)>>::Output>: Send,
{}
```

Prevent lifetime specialization

```
struct MyStruct {
    borrow: Erase<for<'owner> fn(&'owner ()) -> (&'owner String,)>,
    owner: String,
}
```

Not only fixed the soundness hole, but also no need to replace lifetime at all!

Prevent lifetime specialization

```
struct MyStruct {
    borrow: Erase<for<'owner> fn(&'owner ()) -> (&'owner String,)>,
    owner: String,
}
```

Not only fixed the soundness hole, but also no need to replace lifetime at all!

There exists a unstable feature, unsafe binder, which does exactly this.

```
struct MyStruct {
    borrow: unsafe<'owner> &'owner String,
    owner: String,
}
```

Prevent lifetime specialization

```
error: higher-ranked lifetime error
  --> examples/demo.rs:34:5
   |
34 |     assert_send::<Foo>();
   |     ^^^^^^^^^^^^^^^^^^^^
   |
   = note: could not prove `Foo: Send`
```

Prevent lifetime specialization

```
struct MyStruct {
    borrow: Erase<for<'owner> fn(&'owner ()) -> (&'owner String,)>,
    owner: String,
}
```

Sound now?

Prevent lifetime specialization

```
struct MyStruct {
    borrow: Erase<for<'owner> fn(&'owner ()) -> (&'owner String,)>,
    owner: String,
}
```

Sound now?

Almost!

Unpin protection

```
#[repr(transparent)]
struct Borrowed<T>(PhantomPinned, T);

impl<T> Deref for Borrowed<T> { type Target = T; ... }

struct MyStruct {
    borrow: Erase<for<'owner> fn(&'owner ()) -> (&'owner String,)>,
    owner: Borrowed<String>,
}
```

Some future proofing in case pinning becomes per-byte.

Caveat: the wrapper struct needs to be covariant!

Drop order check

This is clearly broken:

```
#[pin_data]
struct MyStruct {
    owner: String,
    borrow: PrintOnDrop<&'owner String>,
}
```

because `owner` is dropped first. Easy to check.

Drop order check

What will happen if someone writes this and try to put `owner` into `print_on_drop` field?

```
#[pin_data]
struct MyStruct {
    borrow: &'owner String,
    owner: String,
    print_on_drop: PrintOnDrop<&'later_owner String>,
    later_owner: String,
}

stack_pin_init!(let x = pin_init!(MyStruct {
    owner: "hello world".to_owned(),
    borrow: owner,
    later_owner: "hello world".to_owned(),
    print_on_drop: PrintOnDrop(owner),
}));
```

Drop order check

```
error: lifetime may not live long enough
  --> examples/demo.rs:74:9
   |
70 |       stack_pin_init!(let x = pin_init!(MyStruct {
   |  _____________________________-
71 | |         owner: "hello world".to_owned(),
72 | |         borrow: owner,
73 | |         later_owner: "hello world".to_owned(),
74 | |         print_on_drop: PrintOnDrop(owner),
   | |         ^^^^^^^^^^^^^ argument requires that `'1` must outlive `'2`
75 | |     }));
   | |      -
   | |      |
   | |______has type `__PinDataLt<'1, '_>`
   |        has type `__PinDataLt<'_, '2>`
   |
   = note
: requirement occurs because of the type `__PinDataLt<'_, '_>`, which makes the generic argument `'_`
invariant
   = note: the struct `__PinDataLt<'owner, 'later_owner>` is invariant over the parameter `'owner`
   = help: see <https://doc.rust-lang.org/nomicon/subtyping.html> for more information about variance
```

Drop order check

```
#[pin_data]
struct MyStruct {
    what_the_heck: &'later_owner &'owner (),
    borrow: &'owner String,
    owner: String,
    print_on_drop: PrintOnDrop<&'later_owner String>,
    later_owner: String,
}
```

Oh no!!!! This now builds!

Drop order check

```
#[pin_data]
struct MyStruct {
    what_the_heck: &'later_owner &'owner (),
    borrow: &'owner String,
    owner: String,
    print_on_drop: PrintOnDrop<&'later_owner String>,
    later_owner: String,
}
```

Oh no!!!! This now builds!

The implied bound will now say `'owner: 'later_owner` and now `&'owner String` → `&'later_owner String` conversion is possible!

And there's no way to prevent the implied bound from happening.

Drop order check

But we can check implied bound:

```
fn __drop_check<'owner, 'later_owner: 'owner>(_: &MyStruct) {
    let what_the_heck: &'later_owner &'owner () = todo!();
    let borrow: &'owner String = todo!();
    let owner: String = todo!();
    let print_on_drop: PrintOnDrop<&'later_owner String> = todo!();
    let later_owner: String = todo!();
}
```

Drop order check

```
error: lifetime may not live long enough
  --> examples/demo.rs:14:20
   |
14 |     what_the_heck: &'later_owner &'owner (),
   |                    ^^^^^^^^^^^^^^^^^^^^^^^^ requires that `'owner` must outlive `'later_owner`
15 |     borrow: &'owner String,
16 |     owner: String,
   |     ----- lifetime `'owner` defined here
17 |     print_on_drop: PrintOnDrop<&'later_owner String>,
18 |     later_owner: String,
   |     ---------- lifetime `'later_owner` defined here
   |
   = help: consider adding the following bound: `'owner: 'later_owner`
```

Accessors

- Direct access to fields are blocked (`Erase`).
- Covariant fields have direct accessor method generated with lifetime replaced with `'this`:
  - `fn field<'this>(&'this self) -> &'this String`
  - Covariance is checked instead of assumed:

```
fn covariance_check<'long: 'short, 'short>(long: Foo<'long>) -> Foo<'short> {
    long
}
```

- Non-covariant fields only have callback based accessor:
  - `fn with_field<'this>(&'this self, f: impl for<'owner> FnOnce(&'this Mutex<&'owner String>) -> R ) -> R`

Accessors

```
error: lifetime may not live long enough
 --> examples/demo.rs:5:14
  |
3 | #[pin_data]
  | ----------- in this attribute macro expansion
4 | struct SelfRef {
5 |     not_cov: Box<dyn Fn(&'str str) -> bool + 'str>,
  |              ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  |              |
  |              lifetime `'__short` defined here
  |              lifetime `'__long` defined here
  |
function was supposed to return data with lifetime `'__long` but it is returning data with lifetime
`'__short`
  |
  = help: consider adding the following bound: `'__short: '__long`
  = note
: this error originates in the attribute macro `pin_data` (in Nightly builds, run with -Z macro-backtrace
for more info)
```

Projections

Traditional projection uses `.project()` without callback syntax:

- Borrowed fields are downgraded to shared reference: `String` becomes `&'this String`
- Covariant self-ref fields are downgraded to shared reference, with lifetime replaced with `'this`: `&'owner String` becomes `&'this &'this String`
- Invariant self-ref fields are removed

New callback projection `.with_project()` mints field lifetime via HRTB:

- Borrowed fields are downgraded to shared reference, with field lifetime: `String` becomes `&'owner String`
- Self-ref fields appear normally. `&'this mut &'owner String`

The soundness journey

This is perhaps one of the most thorough defences ever done by a self-referential library.

- Takes into account of auto traits, lifetime specialization, pinning, variances, wellformedness, implied bounds
- No syntactic checks for soundness, everything translated to semantics and delegated to the Rust type system.
  - Syntactic custom checks still exist for better diagnostics.

Kernel's constraints

- Ergonomic ✅
- No allocation ✅
- Soundness ✅

So now we ticked all 3 boxes, ergonomic, no allocation, and soundness.

Kernel's constraints

- Ergonomic ✅
- No allocation ✅
- Soundness ✅ ❓

So now we ticked all 3 boxes, ergonomic, no allocation, and soundness.

Or is it? Up until early this month, I was still very convinced that this is sound. Until Danilo tried the feature with nova-core, and provide me with a compilation error that he couldn't solve.

A failed build

```
#[pin_data]
pub(crate) struct Cmdq<'a>
{
    #[pin]
    #[not_covariant]
    inner: Mutex<CmdqInner<'a>>,
    ...
}
```

Mutex, causing this to be invariant (we actually won't put data with lifetime in, we only mutate what is *already contained inside*.)

The core part of the code looks like this. In Nvidia GPUs, there is a command queue which all communication with the GSP needs to use. Naturally, we have a mutex here to prevent concurrent access to it. We actually won't put data with lifetime in, we only mutate what is *already contained inside*.

But, the variance rule wouldn't know, so the mutex cause this type to be invariant over `'a`.

A failed build

```
#[pin_data]
struct NovaCore<'bound> {
    reg: auxiliary::Registration<'gpu, ForLt!(NovaCoreApi<'_>)>,
    #[not_covariant] // Invariant-pocalypse
    gpu: Gpu<'bar>,
    bar: Bar<'bound, BAR0_SIZE>,
}

fn probe<'bound>(pdev: &'bound Device<Core<'_>>) ->
    impl PinInit<NovaCore<'bound>, Error> + use<'bound>
{
    try_pin_init!(NovaCore {
        bar: pdev.iomap_region_sized::<BAR0_SIZE>(0, c"nova-core/bar0")?,
        gpu <- Gpu::new(pdev, bar),
        reg: auxiliary::Registration::new_with_lt(
            pdev.as_ref(), c"nova-drm", 0, c"nova-core",
            // This expects `&'gpu Gpu<'gpu>`, but we get `&'gpu Gpu<'bar>`.
            pin_init!(NovaCoreApi { gpu, pdev }),
        )?,
    })
}
```


Let's switch back to the top-level code that I showed earlier. Due to the invariance, it infects all types along the way, and cause the top-level `Gpu` type to be invariant too.

It is fine by itself, but when constructing the auxiliary device, we need to put a reference of the GPU struct into the auxiliary device registration, which uses the `ForLt` pattern. Thus, it can only be generic over a single lifetime.

Due to the invariance, the resulting type would have two lifetimes and you cannot not coerce one to another.

A hack

```
#[pin_data]
pub(crate) struct Cmdq<'a>
{
    #[pin]
    #[not_covariant]
    // Now we can reference our copy, not the original ``'a`.
    inner: Mutex<CmdqInner<'dev>>,
    // Put a copy of device here.
    dev: &'a Device<Bound>,
    // ...
}
```

`inner` makes `'dev` invariant, but the type is still covariant over `'a`. Invariant-pocalypse solved!

I then discovered a hack that can be used to bypass this. First, you stash a copy of the `Device`, which is where the lifetime originates from. Then you can borrow from this field for the lifetime.

Now the type is invariant over `'dev`, but still covariant over `'a`. `'dev` is erased, so we're left with a single, covariant lifetime!

Due to wellformedness, `'a` will necessarily outlive `'dev`, so we can still perform `'a` to `'dev` coercion during construction.

A hack

```
#[pin_data]
pub(crate) struct Cmdq<'a>
{
    #[pin]
    #[not_covariant]
    // Now we can reference our copy, not the original `'a`.
    inner: Mutex<CmdqInner<'dev>>,
    // Put a copy of device here.
    dev: &'a Device<Bound>,
    // ...
}
```

`inner` makes `'dev` invariant, but the type is still covariant over `'a`. Invariant-pocalypse solved!

Is this really sound?

If unsound, why is this unsound?

Does the lifetime `'dev` has some spooky-action-at-a-distance and magically affect the other lifetime too? This interaction do not appear in existing Rust lifetimes.

To understand this, I need to find an abstraction that abstracts away from pin-init.

Existential lifetimes

That abstraction is existential lifetime.

Existential lifetime

Say we have a hypothetical existential lifetime,

```
exists<'a, 'b> Foo<'a, 'b>
```

indicates that there exists some lifetime that the value is `Foo<'a, 'b>`

Operations:

$$\frac{\Gamma, \texttt{'a} : \mathrm{Lt} \vdash e : \texttt{T<'a>}}{\Gamma \vdash \texttt{pack!(e)} : \texttt{exists<'a> T<'a>}}\ \text{Pack}$$

$$\frac{\Gamma \vdash x : \texttt{exists<'a> T<'a>} \qquad \Gamma, \texttt{'a} : \mathrm{Lt}, f : \texttt{T<'a>} \vdash e : R}{\Gamma \vdash \texttt{unpack!(x, |f| e)} : R}\ \text{Unpack}$$

The existential lifetime only have two operations. They are typically called **pack** and **unpack** in type theory literature.

You can see that the `unpack!()` expression is very similar to the `with_project()` or `with_field()` accessors that we generate.

Encoding of pin-init self-references

Consider `Foo<'a>` that has self-references. We can imagine the initialized

```
Pin<&'this mut Foo<'a>>
```

to be of type

```
Pin<&'this mut exists<'field_a, 'field_b> FooLt<'a, 'field_a, 'field_b>>
```

This maps cleanly to the `with_project()`.

Individually

```
struct Foo {
  borrow: exists<'owner> &'owner String,
  owner: String,
}
```

This maps cleanly to the `with_field()` accessor.

This encoding works for owned values too; despite being owned, the slot it lives in still have a lifetime, and you can consider the lifetime that the drop glue receives being the final lifetime of owned values.

Lower bound requirement

If we erase all lifetimes using existential types, then the result becomes lifetime free and thus `'static`!

For each erased lifetime, it must **outlive** some free environment lifetime.

```
type X<'env> = exists<'a: 'env, 'b: 'env> Foo<'a, 'b>
```

Does not break the encoding of pin-init self-reference, because all field lifetimes outlives the struct's `'this` lifetime.

This applies to the individual field expansion encoding too; in such cases, the struct's `'this` lifetime still conceptually exist, just not nameable.

We have existential type at home

Given **Church-Encoding** of existential types,

$$\exists \alpha.B \cong \forall \beta.(\forall \alpha.B \to \beta) \to \beta$$

We could also have the following encoding in Rust:

```
struct ExistFoo<'l>(Box<dyn Fn(&mut dyn for<'a, 'b> FnMut(&'l Foo<'a, 'b>)) + 'l>);

impl<'l> ExistFoo<'l> {
    fn unpack<R>(self, f: impl for<'a, 'b> FnOnce(&'l Foo<'a, 'b>) -> R) -> R {
        let mut f = Some(f);
        let mut r = None;
        self.0(&mut |foo| { r = Some(f.take().unwrap()(foo)); });
        r.unwrap()
    }

    fn pack<'a: 'l, 'b: 'l>(val: &'l Foo<'a, 'b>) -> Self {
        Self(Box::new(move |f| f(val)))
    }
}
```


Fun fact, due to Church-encoding, existential quantification can be implemented via universal quantification.

Without using unsafe!

Perhaps it's not surprising that the way you implement existential lifetime is via existential type (`dyn Trait`).

This is the `ForLt!` turned inside out!

Back to the unsound? case

We can convert the `Cmdq` example to the following existential type:

```
type X<'a> = exists<'b> (&'b &'a String, Mutex<&'b String>);
```

Naively, we would consider the `'a` to be covariant, because if we ignore the existence of `exists`, then `'a` is only in covariance position.

Now the question is about whether this type can be covariant over `'a`.

Back to the unsound? case

However, consider this code

```
type X<'a> = exists<'b> (&'b &'a String, Mutex<&'b String>);

let mut s = "hello world".to_owned();
let r = &s;
let x = pack!(&r, Mutex::new(&r));
{
    let mut s = "hello world".to_owned();
    shorten(x, &s).unpack(|x| { // Shortening covariant `'a` to that of `s`.
      *x.1.lock().unwrap() = &s; // Okay because 'a: 'b
    })
}
x.unpack(|x| {
    println!("{}", *x.1.lock().unwrap()); // UAF
})
```


We only shorten a covariant lifetime `'a` to that of `s`. Due to the implied wellformedness bound `'a` outlives `'b`, so after shortening, the wellformedness bound becomes `'s: 'b`. So we can now put a reference to `s` inside a mutex because it coerces.

When we leave the scope, we will get a UAF.

Back to the unsound? case

Existing analogy that does not use existential lifetime:

```
struct X<'a, 'b> {
    p: &'b &'a (),
    b: Mutex<&'b ()>,
}
```

This type is covariant over `'a` and is sound. Why does it become unsound for existential type?

Back to the unsound? case

Existing analogy that does not use existential lifetime:

```
struct X<'a, 'b> {
    p: &'b &'a (),
    b: Mutex<&'b ()>,
}
```

This type is covariant over `'a` and is sound. Why does it become unsound for existential type?

Answer: `'static` ⟶ `'a` ⟶ `'b`

The invariant `'b` forms a lower bound of `'a`. While `'a` can shorten, it cannot be shortened past `'b`, and this is enforced by the Rust compiler on subtyping sites. With existential lifetime, `'b` is erased, and the lower bound disappeared.

Proper subtyping rule

Therefore, in order for the existential type to be *covariant* over a free lifetime:

- it needs to be in covariant position
- no erased *non-covariant* lifetime may be its lower bound implied by WFness. They *can* be upper bounds.

The same is true for contravariant lifetime, no erased non-contravariant lifetime may be its upper bound.

Proper subtyping rule

Therefore, in order for the existential type to be *covariant* over a free lifetime:

- it needs to be in covariant position
- no erased *non-covariant* lifetime may be its lower bound implied by WFness. They *can* be upper bounds.

Back to the example:

```
type X<'a> = exists<'b> (&'b &'a String, Mutex<&'b String>);
```

As `'b` *is* a lower bound of `'a` for wellformedness, the existential type must be *invariant* over `'a`.

The same is true for contravariant lifetime, no erased non-contravariant lifetime may be its upper bound.

Alternative phrasing is that “the type must also be covariant over any erased lower bound lifetime”.

Existential lifetime does not exist yet in Rust today; but it needs to follow this rule when it does.

Arguably, the unsafe binder types should also follow this rule (it doesn’t today).

The fix in pin-init

The invariant closure rule:

- If a field is not covariant over lifetime of a field it borrows, then the struct must be invariant over the type of the *borrowed* field.
- This applies transitively, so if the borrowed field borrows another field, that field will be made invariant too, even it is itself a covariant borrow.
  - Sorry, no cheating!
  - The accessor generated can still be covariant, it's just that the type will be invariant.

The with projection type can also be covariant, because it has no hidden lifetimes anymore and Rust compiler can enforce bounds.

Back to Nova's case

```
#[pin_data]
pub(crate) struct Cmdq<'a>
{
    #[pin]
    #[not_covariant]
    inner: Mutex<CmdqInner<'a>>,
    ...
}
```

So the hack is blocked and this is invariant over `'a` again. But we want it to be covariant!

Again, we never put any data with `'a` liftime in, we just want to mutate what's already in there.

Back to Nova's case

Proper subtyping rule

Therefore, in order for the existential type to be *covariant* over a free lifetime:

- it needs to be in covariant position
- no erased *non-covariant* lifetime may be its lower bound implied by WFness. They *can* be upper bounds.

The solution

```
#[pin_data]
pub(crate) struct Cmdq<'a>
where
  exists<'cmdq>: 'a,
{
    #[pin]
    #[not_covariant]
    inner: Mutex<CmdqInner<'cmdq>>,
    ...
}
```

Here, `'cmdq` is an upper bound. Shortening `'a` preserves the bound.

This is implemented and is actually working code. With this, all lifetime related unsafe removed from Nova (except debugfs, which can also be made safe by introducing registration data to bus devices).

Implementation-wise, it is almost identical to the field lifetimes; almost all the infrastructure is shared. They are almost always introduced using HRTB, the only exception is that in constructor, they are lifetime parameter. 200 LoC implementation on top of self-ref series, mostly parsing.

Next steps

- Work out a general existential lifetime solution as a building block?
- Upstream to the kernel
  - MVP targeting this cycle.
  - Shared borrow only; no explicit annotation; all fields to be covariant.
- Keep improving the diagnostics
  - Maybe even do some custom diagnostics for it in klint?
- Motivate language design?

Thank you for listening

Thank you for listening
—