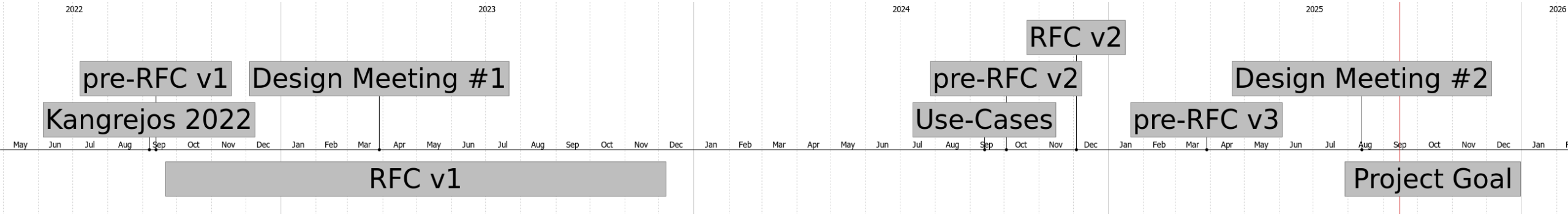
Field Projections

Benno Lossin

History of Field Projections



What are Field Projections?

```
struct Struct {
    field: Field,
}

fn project(r: &MyStruct) -> &Field {
    &r.field
}
```

Raw Pointers

```
unsafe fn project_raw(
    r: *mut Struct,
) -> *mut Field {
    unsafe { &raw mut (*r).field }
}
```

`MaybeUninit<T>`

```
fn project_uninit(
    r: &mut MaybeUninit<Struct>,
) -> &mut MaybeUninit<Field> {
    todo!("possible, but verbose")
}
```

`NonNull<T>`

```
fn project_non_null(ptr: NonNull<MyStruct>) -> NonNull<Foo>;
```

`VolatilePtr<T>`

```
VolatilePtr<Struct> -> VolatilePtr<Field>
```

`Untrusted<T>`

```
&mut Untrusted<Struct> -> &mut Untrusted<Field>

&Untrusted<Struct> -> &Untrusted<Field>
```

Container-Projections

All of the projections that we have seen so far have the following shape:

```
Container<'a, Struct> -> Container<'a, Field>
```

But there are also more complicated projections that change the output of the projection.

Complex Projections

In general, a projection can have the following shape:

```
Container<'a, Struct> -> Output<'a, Field>
```

Where `Output` is allowed to depend on the concrete field of the struct.

`Pin<&mut T>`

```
struct MyStruct {
    foo: Foo,
    #[pin]
    bar: Bar,
}

fn project_foo_pin(_: Pin<&mut MyStruct>) -> &mut Foo;

fn project_bar_pin(_: Pin<&mut MyStruct>) -> Pin<&mut Bar>;
```

RCU

- RCU is a special locking mechanism in the kernel,
- RCU can only protect pointers,
- RCU protects readers from writers by only changing pointers atomically and waiting for readers to finish before destroying the allocations,
- RCU needs to be combine with another locking mechanism to synchronize between multiple writers.

RCU

```
struct Data {
    // RCU-protected data must be stored in a pointer type wrapped by RCU.
    // internally, this is just an `AtomicPtr<Config>`
    #[pin]
    cfg: Rcu<Box<Config>>,
    // Data that isn't protected by RCU is just stored normally.
    other: i32,
}

struct Config {
    size: usize,
    name: &'static CStr,
}
```

We store `Arc<Mutex<Data>>` somewhere in our module & then access it from the driver.

RCU (Read)

```
fn size(data: &rfl::Mutex<Data>) -> usize {
    // `&Mutex<T>` allows projecting to fields of type `Rcu<U>`,
    // but not to other fields (that would be unsound).
    let cfg: &Rcu<Box<Config>> = proj!(data.cfg);
    // now we begin the critical read section of RCU.
    let rcu = rcu::read_lock();
    let cfg: &Config = cfg.get(&rcu);
    cf.size
}
```

RCU (Write)

```
fn set_config(data: &rfl::Mutex<Data>, config: Config) {
    let mut data: MutexGuard<'_, Data> = data.lock();
    // Normal data can just be handled as usual using field projections.
    // (note: the mutex pins its data, so we need projections here)
    *proj!(data.other) = 42;

    // Maybe somewhat surprisingly, we can obtain a `Pin<&mut Rcu<...>>`:
    let cfg: Pin<&mut Rcu<Box<Config>>> = proj!(data.cfg);

    // `Rcu::set` has `Pin<&mut Rcu>` as the receiver, so it can only be called,
    // if the external lock is taken.
    let _old = cfg.set(Box::new(config));

    // When `_old` is dropped, `synchronize_rcu` is executed, waiting for a
    // grace period to end (ie all currently active critical read sections
    // must end). This guarantees that any readers still holding onto a
    // pointer to the contents of `_old` have a valid pointer.
    drop(_old);
}
```

Field Projection Operator

Have a new operator for field projections similar to reborrowing references:

```
fn project(r: &mut MaybeUninit<Struct>) -> &mut MaybeUninit<Field> {
    @mut r->field
}
```

Lot's of different syntaxes being discussed:

- `@r->field` and `@mut r->field`
- `r.ref.@field` and `r.@field`
- `r.@field.ref` and `r.@field`
- `@r~field` and `@mut r~field`
- `r.@field` and `r.mut@field`
- just `r.@field`

And many more sigil options.

Projections Beyond `struct`

We can easily extend field projections to:

- tuples
- arrays

If we add some more features we can also support:

- `union`
- `enum`

How to Help

You can help with:

- Motivation
  - can existing applications of field projection be useful in your area of code?
  - do you possibly have additional applications?
- Experimentation
  - when the lang experiment is ready (check status of rust-lang/rust#146307), try out `#![feature(field_projections)]` & give feedback

Questions & Discussion