

Initialization in Rust with `pin-init`

Working towards a chapter in the RfL book about initialization

Benno Lossin <lossin@kernel.org>

Rust for Linux Core Team

Goals of this Talk

- Give an accessible explanation of **why** we need a library *just* for initialization in Rust.
- Provide an up-to-date introduction to initialization in Rust with `pin-init`.
- Obtain information about what helps people understand `pin-init`.
 - Using this, write a chapter for the RfL book.
- Give an outlook on the future of `pin-init` & in-place initialization in Rust.

Questions welcome, but please keep discussions for later.

Why is Initialization so complicated in Rust?

What happens in C? vs. What is usually done in Rust?

```
1 typedef struct {  
2     u8 data[1024 * 1024];  
3 } buf;  
4  
5 void buf_init(struct buf *p, u8 val)  
6 {  
7     memset(buf->data, val, 1024 * 1024);  
8 }
```

out-pointer

C

```
1 struct Buf {  
2     data: [u8; 1024 * 1024],  
3 }  
4  
5 impl Buf {  
6     fn new(val: u8) -> Buf {  
7         Buf { data: [val; 1024 * 1024] }  
8     }  
9 }
```

by-value

overflows the stack!

Rust

Out-Pointer *Conventions* in C

Can be broken in some cases!

```
1 void buf_init(struct buf *p, u8 val)
2 {
3     memset(buf->data, val, 1024 * 1024);
4 }
```

- Using out-pointers requires the function to *always* initialize the out-pointer value.
- Except when they support errors. In the error path, no data should be stored in the out-pointer that needs to be cleaned up.
- Except...

Inherently in conflict with safe Rust:

Cannot be broken!

All safely accessible data in Rust must be initialized.


- Out-pointers don't exist in Rust, they would have to:
 - *Somehow* ensure initialization before returning from the function,
 - *Except* in the error path, where they ensure that they carry no initialized data.

This is difficult!

A more advanced Example in C

```
1  typedef struct {
2      struct mutex lock;
3      struct buf buffer;
4  } my_driver_data;
5
6  int my_driver_probe(struct foo_device *dev)
7  {
8      dev->private = kmalloc(sizeof(struct my_driver_data), GFP_KERNEL);
9      if (!dev->private) {
10         return -ENOMEM;
11     }
12     struct my_driver_data *ptr = (struct my_driver_data *) dev->private;
13     mutex_init(&ptr->lock);
14     buf_init(&ptr->buffer, 0);
15     return 0;
16 }
```

private data is out-pointer



A mutex must not be **moved** after initializing it.

- In Rust, we say that we *pin* a value when we require it to never move again.
- Thus, a mutex is *pinned* at creation.

How does Initialization look like with `pin-init`?

A Basic Example of pin-init

```
1 struct Buf {
2     data: [u8; 1024 * 1024],
3     count: usize,
4 }
5
6 impl Buf {
7     fn new(val: u8) -> impl Init<Self> {
8         init!(Self {
9             data <- init_array_from_fn(|_| val),
10            count: 42,
11        })
12     }
13 }
```



The Details of `pin-init`

- `Init<T>` trait is an *initializer*:
 - It is an object that carries the data needed to initialize a `T`.
 - When it isn't run, no data is initialized.
- Several ways to create an initializer:
 - Any value `T` can be turned into an `Init<T>`.
 - The `init!` macro creates an initializer for a struct given initializers for each field.
 - Several functions in `pin-init` return initializers (for example `zeroed` for zeroable data).
- Need all of this again with slight variations for `pin`:
 - `PinInit` trait, which requires the value to be *pinned* after initialization.
 - `pin_init!` macro, which allows composing `PinInit` for structs.
 - *Additionally*: compatibility with `Init`, since any initializer also is a pin-initializer.

Converting the Mutex Example

```
1  #[pin_data] ← Required for using pin_init! with this struct
2  struct MyDriverData {
3      #[pin] ← Marks the buffer field as structurally pinned
4      buffer: Mutex<Buf>,
5  }
6
7  impl FooDriver for MyDriver {
8      type Data = MyDriverData;
9
10     fn probe(dev: ARef<FooDevice>) -> impl PinInit<Self::Data> {
11         pin_init!(MyDriverData {
12             buffer <- Mutex::new(Buf::new(0)),
13         })
14     }
15 }
```



Error Handling

```
1  #[pin_data]
2  struct MyDriverData {
3      shared_data: Arc<SharedData>,
4      #[pin]
5      buffer: Mutex<Buf>,
6  }
7
8  impl MyDriverData {
9      fn new() -> impl PinInit<Self, Error> {
10         pin_init!(MyDriverData {
11             shared_data: Arc::new(SharedData::new(), GFP_KERNEL)?,
12             buffer <- Mutex::new(Buf::new(0)),
13         })
14     }
15 }
```

Rust

Arc creation is fallible, since it allocates

Can use ? operator which short-circuits on error

New Features and Language Integration

New features added over the last two years:

- Code blocks to run arbitrary code before, between, and after initializing fields,
- Access to previously initialized fields in code blocks and later fields,
- Pin projections to convert `Pin<&mut Struct>` into `Pin<&mut Field>` or `&mut Field`,
- `[pin_]init_scope` function to compute data before creating the initializer and pass it to multiple fields.

Feature: Code Blocks & Access to Initialized Fields

```
1 struct MyStruct {  
2     field1: i32,  
3     field2: u32,  
4 }  
5  
6 fn foo();  
7 fn bar(x: &mut i32);  
8 fn baz(x: &mut i32, y: &mut u32);  
9 fn bam();
```

```
1 init!(MyStruct {  
2     _: {  
3         foo();  
4     },  
5     field1: 42,  
6     _: {  
7         foo();  
8         bar(field1);  
9     },  
10    field2: 24,  
11    _: {  
12        baz(field1, field2);  
13    },  
14    _: {  
15        bam();  
16    },  
17 })
```


- Sometimes one needs to pass data to multiple fields at the same time,
- `fn [pin_]init_scope(...)` serves this purpose:

```
1 init_scope(|| {  
2     let foo = Foo::new();  
3     init!(Bar {  
4         bar: foo.bar.clone(),  
5         foo,  
6     })  
7 })
```



Pin Projections

```
1 #[pin_data]
2 struct Struct {
3     field1: usize,
4     #[pin]
5     field2: Mutex<usize>,
6 }
```

 Rust

Auto-generated by `#[pin_data]`:

```
1 struct PinProjectedStruct<'a> {
2     field1: &'a mut usize,
3     field2: Pin<&'a mut Mutex<usize>>,
4 }
5
6 impl Struct {
7     fn project(self: Pin<&mut Self>) -> PinProjectedStruct<'_> {
8         // Uses unsafe code to perform the correct projections.
9     }
10 }
```

 Rust

Upcoming Features of `pin-init`

- Many improvements coming to `pin-init` due to more usage (especially in DRM & by Danilo, many thanks!).
 - This is great, keep the suggestions coming!
- Big improvement *behind the scenes*: `syn` to parse Rust code instead of declarative macros. This allows for:
 - Better error reporting and better syntax support,
 - Comfortable maintenance (Gary Guo will be co-maintaining the library when it switches to `syn`),
 - Easier extensions of the existing syntax.
- Closure syntax that improves upon `init_scope`,
- Compatibility with normal Rust syntax & `rustfmt` support.
- More customizability for the `init!` macro (default error, should previous fields be accessible).

- Language integration has been started,
- Pin ergonomics make generated pin-projections obsolete,
- Project goal by Alice Ryhl & Taylor Cramer on in-place initialization to implement `pin-init` as a language feature.
 - Discussion by many different people,
 - Many, *many* more considerations for better compatibility with existing features (`async`, `try`, classical control-flow)

Thank You for Your Attention!

Questions & Discussion
