Safe Pinned Initialization in Rust

Benno Lossin (y86-dev@proton.me)

September 7, 2022

Overview

Why Pinned Initialization is a Problem

- pinning requires **`unsafe`** a *lot*

Why Pinned Initialization is a Problem

- pinning requires **`unsafe`** a *lot*
- safe pin-projection already solved by `pin-project` (without proc-macros by `pin-project-lite`)

Why Pinned Initialization is a Problem

- pinning requires **`unsafe`** a *lot*
- safe pin-projection already solved by `pin-project` (without proc-macros by `pin-project-lite`)
- but that is not the only use of **`unsafe`**:

Why Pinned Initialization is a Problem

- pinning requires **unsafe** a *lot*
- safe pin-projection already solved by `pin-project`
- but that is not the only use of **unsafe**:

```rust
pub struct SelfReferential {
    value: u32,
    ptr: *const u32,
    _pin: PhantomPinned,
}

impl SelfReferential {
    /// # Safety
    /// The caller guarantees to call `init`
    /// before they use the returned value.
    pub unsafe fn new(value: u32) -> Self { ... }
}
```

Why Pinned Initialization is a Problem

samples/rust/rust_miscdev.rs

```rust
let mut state = Pin::from(UniqueRef::try_new(Self {
    // SAFETY: `condvar_init!` is called below.
    state_changed: unsafe { CondVar::new() },
    // SAFETY: `mutex_init!` is called below.
    inner: unsafe { Mutex::new(SharedStateInner { token_count: 0 }) },
})?);

// SAFETY: `state_changed` is pinned when `state` is.
let pinned = unsafe {
    state.as_mut().map_unchecked_mut(|s| &mut s.state_changed)
};
kernel::condvar_init!(pinned, "SharedState::state_changed");

// SAFETY: `inner` is pinned when `state` is.
let pinned = unsafe {
    state.as_mut().map_unchecked_mut(|s| &mut s.inner)
};
kernel::mutex_init!(pinned, "SharedState::inner");

Ok(state.into())
```

My Initial Attempt at a Solution

- type tracks initialization via const generics:

My Initial Attempt at a Solution

- type tracks initialization via const generics: transmute pointer with an uninitialized pointee `Struct<Init = false>` → `Struct<Init = true>` after initialization

My Initial Attempt at a Solution

- type tracks initialization via const generics: transmute pointer with an uninitialized pointee `Struct<Init = false>` $\rightarrow$ `Struct<Init = true>` after initialization
- relying on proc-macros to insert these generics and functions:

My Initial Attempt at a Solution

- type tracks initialization via const generics: transmute pointer with an uninitialized pointee `Struct<Init = false>` $\rightarrow$ `Struct<Init = true>` after initialization
- relying on proc-macros to insert these generics and functions:
  - each field of `Struct` gets `Init` added

My Initial Attempt at a Solution

- type tracks initialization via const generics: transmute pointer with an uninitialized pointee `Struct<Init = false>` $\rightarrow$ `Struct<Init = true>` after initialization
- relying on proc-macros to insert these generics and functions:
  - each field of `Struct` gets `Init` added
  - `init()` function calls the init function for each field

My Initial Attempt at a Solution

- type tracks initialization via const generics: transmute pointer with an uninitialized pointee `Struct<Init = false>` $\rightarrow$ `Struct<Init = true>` after initialization
- relying on proc-macros to insert these generics and functions:
  - each field of `Struct` gets `Init` added
  - `init()` function calls the init function for each field
- overall very complex, especially the initialization code flow

My Initial Attempt at a Solution

- type tracks initialization via const generics: transmute pointer with an uninitialized pointee `Struct<Init = false>` → `Struct<Init = true>` after initialization
- relying on proc-macros to insert these generics and functions:
  - each field of `Struct` gets `Init` added
  - `init()` function calls the init function for each field
- overall very complex, especially the initialization code flow
- tried to solve a bigger issue: storing partially initialized data

My Initial Attempt at a Solution

- type tracks initialization via const generics: transmute pointer with an uninitialized pointee `Struct<Init = false>` → `Struct<Init = true>` after initialization
- relying on proc-macros to insert these generics and functions:
  - each field of `Struct` gets `Init` added
  - `init()` function calls the init function for each field
- overall very complex, especially the initialization code flow
- tried to solve a bigger issue: storing partially initialized data
- on top of that it is *unsound*

Current Solution

Current Solution

> *"I disagree. There are things that should not be proc-macros – at all – and I hope Rust does not make the mistake of leaving everything up to a proc-macro. For instance, I hope* `let else` *is not rejected just because it could be done as a proc macro."*
>
> *—Miguel Ojeda, Github Issue 772*

Current Solution

Current Solution

central idea: use a struct initializer!

```
struct MyStruct {
    a: u32,
    b: u64,
    c: usize,
}

let my_struct = MyStruct {
        a: todo!(),
        b: todo!(),
        b: todo!(),
```

Current Solution

central idea: use a struct initializer!

```
struct MyStruct {
    a: u32,
    b: u64,
    c: usize,
}

let my_struct = MyStruct {
        a: todo!(),
        b: todo!(),
        b: todo!(),
//      ^ used more than once
};
// ^^ missing `c` in initializer
```

Current Solution

central idea: use a struct initializer!

```
struct MyStruct {
    a: u32,
    b: u64,
    c: usize,
}

let my_struct = MyStruct {
        a: todo!(),
        b: todo!(),
        b: todo!(),
//      ^ used more than once
};
// ^^ missing `c` in initializer
```

the compiler can already do everything we need!

Current Solution

The new API provides a macro to initialize structs:

```
1  struct MyStruct {
2      a: u32,
3      b: u64,
4      c: usize,
5  }
6
7  let my_struct: Pin<Box<MaybeUninit<MyStruct>>> =
8      Box::pin(MaybeUninit::uninit());
```

Current Solution

The new API provides a macro to initialize structs:

```
1  struct MyStruct {
2      a: u32,
3      b: u64,
4      c: usize,
5  }
6
7  let my_struct: Pin<Box<MaybeUninit<MyStruct>>> =
8      Box::pin(MaybeUninit::uninit());
9  let my_struct: Pin<Box<MyStruct>> =
10     init! { my_struct => MyStruct {
11
12
13
14     }
15 };
```

Current Solution

The new API provides a macro to initialize structs:

```
1  struct MyStruct {
2      a: u32,
3      b: u64,
4      c: usize,
5  }
6
7  let my_struct: Pin<Box<MaybeUninit<MyStruct>>> =
8      Box::pin(MaybeUninit::uninit());
9  let my_struct: Pin<Box<MyStruct>> =
10     init! { my_struct => MyStruct {
11         .a = 42;
12         .b = 84;
13         .c = 0;
14     }
15 };
```

Current Solution

The new API provides a macro to initialize structs:

```
1  struct MyStruct {
2      a: Mutex<u32>,
3      b: Semaphore<u64>,
4      c: Custom<usize>,
5  }
6
7  let my_struct: Pin<Box<MaybeUninit<MyStruct>>> =
8      Box::pin(MaybeUninit::uninit());
9  let struct_init = init! { my_struct => MyStruct {
10
11
12
13
14
15 }};
```

Current Solution

The new API provides a macro to initialize structs:

```
1  struct MyStruct {
2      a: Mutex<u32>,
3      b: Semaphore<u64>,
4      c: Custom<usize>,
5  }
6
7  let my_struct: Pin<Box<MaybeUninit<MyStruct>>> =
8      Box::pin(MaybeUninit::uninit());
9  let struct_init = init! { my_struct => MyStruct {
10     Mutex::init(.a, 77);
11
12
13
14
15 }};
```

Current Solution

The new API provides a macro to initialize structs:

```
1  struct MyStruct {
2      a: Mutex<u32>,
3      b: Semaphore<u64>,
4      c: Custom<usize>,
5  }
6
7  let my_struct: Pin<Box<MaybeUninit<MyStruct>>> =
8      Box::pin(MaybeUninit::uninit());
9  let struct_init = init! { my_struct => MyStruct {
10     Mutex::init(.a, 77);
11     init_semaphore!(.b, 42);
12
13
14
15 }};
```

Current Solution

The new API provides a macro to initialize structs:

```
1  struct MyStruct {
2      a: Mutex<u32>,
3      b: Semaphore<u64>,
4      c: Custom<usize>,
5  }
6
7  let my_struct: Pin<Box<MaybeUninit<MyStruct>>> =
8      Box::pin(MaybeUninit::uninit());
9  let struct_init = init! { my_struct => MyStruct {
10     Mutex::init(.a, 77);
11     init_semaphore!(.b);
12     ~let (yay, pattern) = unsafe {
13         crazy_custom_init(.c, "magic?!").await
14     }?;
15 }};
```

Under the Hood

```
1  struct MyStruct {
2      a: u32,
3      b: u64,
4      c: usize,
5  }
6
7  let my_struct: Pin<Box<MaybeUninit<MyStruct>>> =
8      Box::pin(MaybeUninit::uninit());
9
10 let my_struct: Pin<Box<MyStruct>> =
11     init! { my_struct => MyStruct {
12         .a = 42;
13         .b = 84;
14         .c = 0;
15     }
16 };
```

Under the Hood

`.a = 42;` expands to:

Under the Hood

`.a = 42;` expands to:

```
unsafe {
    ptr::write(
        mem::addr_of_mut!(




}
```

Under the Hood

`.a = 42;` expands to:

```
unsafe {
    ptr::write(
        mem::addr_of_mut!(
            (*init::place::PartialInitPlace::___as_mut_ptr(
                &mut my_struct,
                &(|_: &MyStruct| {}),
            )).a
        ),


}
```

Under the Hood

`.a = 42;` expands to:

```
unsafe {
    ptr::write(
        mem::addr_of_mut!(
            (*init::place::PartialInitPlace::___as_mut_ptr(
                &mut my_struct,
                &(|_: &MyStruct| {}),
            )).a
        ),
        42,
    );
}
```

Under the Hood

At the end we want to call

```
unsafe {
    init::place::PartialInitPlace::___assume_init(my_struct)
}
```

Under the Hood

Macro remembers each field that is initialized and then adds generates this:

```
let ___check_all_fields_init = || {
    let _struct: MyStruct = MyStruct {
        a: panic!(),
        b: panic!(),
        c: panic!(),
    };
};
unsafe {
    // SAFETY: all fields have been initialized, or
    // a compile error exists.
    init::place::PartialInitPlace::___assume_init(my_struct)
}
```

Immediate Initialization

one often writes

```
let my_struct = Box::pin(MaybeUninit::uninit());
```

and then invokes `init!` so I created a shortcut:

Immediate Initialization

one often writes

```
let my_struct = Box::pin(MaybeUninit::uninit());
```

and then invokes `init!` so I created a shortcut:

```
let my_struct = init! { @Pin<Box<MyStruct>> => MyStruct {
    .a = 42;
    .b = 84;
    .c = 0;
}};
```

Immediate Initialization

one often writes

```
let my_struct = Box::pin(MaybeUninit::uninit());
```

and then invokes `init!` so I created a shortcut:

```
let my_struct = init! { @Pin<Box<MyStruct>> => MyStruct {
    .a = 42;
    .b = 84;
    .c = 0;
}};
```

this essentially calls

```
let my_struct = Box::pin(MaybeUninit::uninit());
let my_struct = init! { my_struct => MyStruct {
    .a = 42;
    .b = 84;
    .c = 0;
}}?;
```

Immediate Initialization

one often writes

```
let my_struct = Box::pin(MaybeUninit::uninit());
```

and then invokes `init!` so I created a shortcut:

```
let my_struct = init! { @Pin<Box<MyStruct>> => MyStruct {
    .a = 42;
    .b = 84;
    .c = 0;
}};
```

It is implemented for:

- `Box<T>`
- `Ref<T>`
- `UniqueRef<T>`

As well as `Pin<P>` where `P` is any of the above.

Init-Functions

We already introduced init functions before:

```
1  struct MyStruct {
2      mutex: Mutex<u32>,
3  }
4
5  let struct_init = init! { my_struct => MyStruct {
6      Mutex::init(.mutex, 77);
7  }}
```

But we did not specify how to declare one!

Init-Functions

Declaring an init-function:

```
1  struct MyStruct {
2      mutex: Mutex<u32>,
3  }
4
5  impl MyStruct {
6      pub fn init(
7          this: PinInitMe<'_, Self>
8      ) {
9          init! { this => Self {
10             Mutex::init(.mutex, 77);
11         }}
12     }
13 }
```

Init-Functions

Declaring an init-function:

```
1  struct MyStruct {
2      mutex: Mutex<u32>,
3  }
4
5  impl MyStruct {
6      pub fn init<G: Guard>(
7          this: PinInitMe<'_, Self, G>
8      ) -> InitProof<(), G> {
9          init! { this => Self {
10             Mutex::init(.mutex, 77);
11         }}
12     }
13 }
```

Limitations

- init-functions can only initialize a singular field

Limitations

- init-functions can only initialize a singular field
- need to specify structurally pinned fields via `pin_data!` which does not fully support the complete struct syntax

Limitations

- init-functions can only initialize a singular field
- need to specify structurally pinned fields via `pin_data!` which does not fully support the complete struct syntax
- you cannot partially initialize the struct using `new()`, everything has to be done in `init!`

Limitations

- init-functions can only initialize a singular field
- need to specify structurally pinned fields via `pin_data!` which does not fully support the complete struct syntax
- you cannot partially initialize the struct using `new()`, everything has to be done in `init!`
- you cannot combine normal rust statements with the special init statements

Limitations

- init-functions can only initialize a singular field
- need to specify structurally pinned fields via `pin_data!` which does not fully support the complete struct syntax
- you cannot partially initialize the struct using `new()`, everything has to be done in `init!`
- you cannot combine normal rust statements with the special init statements
- When a panic/error occurs inside `init!`, then the fields initialized until then are leaked.

How does this look like in the Kernel?

`samples/rust/rust_miscdev.rs` (before):

```
let mut state = Pin::from(UniqueRef::try_new(Self {
    // SAFETY: `condvar_init!` is called below.
    state_changed: unsafe { CondVar::new() },
    // SAFETY: `mutex_init!` is called below.
    inner: unsafe { Mutex::new(SharedStateInner { token_count: 0 }) },
})?);

// SAFETY: `state_changed` is pinned when `state` is.
let pinned = unsafe {
    state.as_mut().map_unchecked_mut(|s| &mut s.state_changed)
};
kernel::condvar_init!(pinned, "SharedState::state_changed");

// SAFETY: `inner` is pinned when `state` is.
let pinned = unsafe {
    state.as_mut().map_unchecked_mut(|s| &mut s.inner)
};
kernel::mutex_init!(pinned, "SharedState::inner");

Ok(state.into())
```

How does this look like in the Kernel?

samples/rust/rust_miscdev.rs (after):

```
1  init! { @Pin<Ref<Self>> => Self {
2    condvar_init!(.state_changed, "SharedState::state_changed");
3    mutex_init!(
4      .inner,
5      "SharedState::inner",
6      SharedStateInner { token_count: 0 }
7    );
8  }}
```

How does this look like in the Kernel?

`samples/rust/rust_semaphore.rs` (before):

```
let mut sema = Pin::from(UniqueRef::try_new(Semaphore {
    // SAFETY: `condvar_init!` is called below.
    changed: unsafe { CondVar::new() },

    // SAFETY: `mutex_init!` is called below.
    inner: unsafe {
        Mutex::new(SemaphoreInner {
            count: 0,
            max_seen: 0,
        })
    },
})?);

// SAFETY: `changed` is pinned when `sema` is.
let pinned = unsafe { sema.as_mut().map_unchecked_mut(|s| &mut s.changed) };
condvar_init!(pinned, "Semaphore::changed");

// SAFETY: `inner` is pinned when `sema` is.
let pinned = unsafe { sema.as_mut().map_unchecked_mut(|s| &mut s.inner) };
mutex_init!(pinned, "Semaphore::inner");
```

How does this look like in the Kernel?

samples/rust/rust_semaphore.rs (after):

```
let sema = init! { @Pin<UniqueRef<Semaphore>> =>
    Semaphore {
        condvar_init!(.changed, "Semaphore::changed");
        mutex_init!(
            .inner,
            "Semaphore::inner",
            SemaphoreInner {
                count: 0,
                max_seen: 0,
            }
        );
    }
}?;
```

Outlook

- `pin-project-lite` support

Outlook

- `pin-project-lite` support
- formal verification?

Outlook

- `pin-project-lite` support
- formal verification?
- at the moment arbitrary expressions and statements are allowed, is this unsound?

Discussion

- follow development at `https://github.com/y86-dev/simple-safe-init`
- integration into the kernel at `https://github.com/y86-dev/linux`
- please give feedback/discuss syntax/wanted features/other remarks