

pin-init: safe initialisation of pinned structs

Xuan Guo (Gary)

Department of Computer Science and Technology, University of Cambridge, UK

What is the problem with pinning?

- ▶ *Once pinned*, it will be pinned forever until the destructor of T is called.

What is the problem with pinning?

- ▶ *Once pinned*, it will be pinned forever until the destructor of T is called.
- ▶ How about the time *before* the value is pinned?

What is the problem with pinning?

- ▶ *Once pinned*, it will be pinned forever until the destructor of T is called.
- ▶ How about the time *before* the value is pinned?
- ▶ Pinning is originally designed for async Futures, which does not create self-references until the first poll.

What is the problem with pinning?

- ▶ *Once pinned*, it will be pinned forever until the destructor of T is called.
- ▶ How about the time *before* the value is pinned?
- ▶ Pinning is originally designed for async Futures, which does not create self-references until the first poll.
- ▶ We can create the value in an uninitialised state, and pass in a `Pin<&mut T>` to initialise it. This requires unsafe.

What is the problem with pinning?

- ▶ *Once pinned*, it will be pinned forever until the destructor of T is called.
- ▶ How about the time *before* the value is pinned?
- ▶ Pinning is originally designed for async Futures, which does not create self-references until the first poll.
- ▶ We can create the value in an uninitialised state, and pass in a `Pin<&mut T>` to initialise it. This requires unsafe.
- ▶ We can lazily initialise a struct upon first usage. This has additional overhead.

What is the problem with pinning?

- ▶ *Once pinned*, it will be pinned forever until the destructor of T is called.
- ▶ How about the time *before* the value is pinned?
- ▶ Pinning is originally designed for async Futures, which does not create self-references until the first poll.
- ▶ We can create the value in an uninitialised state, and pass in a `Pin<&mut T>` to initialise it. This requires unsafe.
- ▶ We can lazily initialise a struct upon first usage. This has additional overhead.
- ▶ We can provide abstraction for self-referential structs and always box them internally. This requires memory allocation.

What do we want?

- ▶ Safety. We should be able to create and use such pinned type without unsafe. (Obviously the pinned type themselves are still unsafe to implement).

What do we want?

- ▶ Safety. We should be able to create and use such pinned type without unsafe. (Obviously the pinned type themselves are still unsafe to implement).
- ▶ Zero-cost. The abstraction provided should be able to be optimised away and leave no runtime cost.

What do we want?

- ▶ Safety. We should be able to create and use such pinned type without unsafe. (Obviously the pinned type themselves are still unsafe to implement).
- ▶ Zero-cost. The abstraction provided should be able to be optimised away and leave no runtime cost.
- ▶ No Implicit Allocation. Allocation should not be required during initialisation. User should be able to dictate whether it's initialised in a box or on the stack.

What do we want?

- ▶ Safety. We should be able to create and use such pinned type without unsafe. (Obviously the pinned type themselves are still unsafe to implement).
- ▶ Zero-cost. The abstraction provided should be able to be optimised away and leave no runtime cost.
- ▶ No Implicit Allocation. Allocation should not be required during initialisation. User should be able to dictate whether it's initialised in a box or on the stack.
- ▶ Aggregatable. A struct containing multiple pinned types can be safely created and initialised together.

What do we want?

- ▶ Safety. We should be able to create and use such pinned type without unsafe. (Obviously the pinned type themselves are still unsafe to implement).
- ▶ Zero-cost. The abstraction provided should be able to be optimised away and leave no runtime cost.
- ▶ No Implicit Allocation. Allocation should not be required during initialisation. User should be able to dictate whether it's initialised in a box or on the stack.
- ▶ Aggregatable. A struct containing multiple pinned types can be safely created and initialised together.
- ▶ Ergonomics. The abstraction should not be too different from normal Rust.

What do we want?

- ▶ Safety. We should be able to create and use such pinned type without unsafe. (Obviously the pinned type themselves are still unsafe to implement).
- ▶ Zero-cost. The abstraction provided should be able to be optimised away and leave no runtime cost.
- ▶ No Implicit Allocation. Allocation should not be required during initialisation. User should be able to dictate whether it's initialised in a box or on the stack.
- ▶ Aggregatable. A struct containing multiple pinned types can be safely created and initialised together.
- ▶ Ergonomics. The abstraction should not be too different from normal Rust.
- ▶ Fallible. No assumption is made about success of initialisation.

Starting point

```
impl RawMutex {  
    // Unsafe because user needs to be initialise it before use  
    unsafe fn uninit() -> Self;  
    // Unsafe because it cannot be initialised twice  
    unsafe fn init(self: Pin<&mut Self>);  
}
```

Starting point

```
impl RawMutex {  
    // Unsafe because user needs to be initialise it before use  
    unsafe fn uninit() -> Self;  
    // Unsafe because it cannot be initialised twice  
    unsafe fn init(self: Pin<&mut Self>);  
}
```

Problem: We don't want the type to have a dedicated uninitialised state.

Starting point

```
impl RawMutex {  
    // Unsafe because user needs to be initialise it before use  
    unsafe fn uninit() -> Self;  
    // Unsafe because it cannot be initialised twice  
    unsafe fn init(self: Pin<&mut Self>);  
}
```

Problem: We don't want the type to have a dedicated uninitialised state. Does MaybeUninit work?

MaybeUninit

```
impl RawMutex {  
    // Caller must treat this as `Pin<&mut Self>` after returning and respecting drop  
    ↪ guarantee.  
    unsafe fn init(this: Pin<&mut MaybeUninit<Self>>);  
}
```

MaybeUninit

```
impl RawMutex {  
    // Caller must treat this as `Pin<&mut Self>` after returning and respecting drop  
    ↪ guarantee.  
    unsafe fn init(this: Pin<&mut MaybeUninit<Self>>);  
}
```

Problem: init function still unsafe to call.

Abstraction

```
struct PinUninit<'a, T> { ... }
```

```
impl<'a, T> PinUninit<'a, T> {
```

```
    // Creator must call an initialiser, and treat `ptr` as `Pin<&mut Self>` after  
    ↪ it is being initialised.
```

```
    unsafe fn new(ptr: &'a mut MaybeUninit<T>) -> Self;  
}
```

```
impl RawMutex {
```

```
    fn init(this: PinUninit<'_, Self>);  
}
```

Abstraction

```
struct PinUninit<'a, T> { ... }

impl<'a, T> PinUninit<'a, T> {
    // Creator must call an initialiser, and treat `ptr` as `Pin<&mut Self>` after
    ↪ it is being initialised.
    unsafe fn new(ptr: &'a mut MaybeUninit<T>) -> Self;
}

impl RawMutex {
    fn init(this: PinUninit<'_, Self>);
}
```

Problem: this is unsound as there is no guarantee that 'init' actually initialises. We want the init function to be unsafe to define but safe to call.

Abstraction

```
struct PinUninit<'a, T> { ... }
```

```
// Unsafe to create token indicating that indeed something is initialised.
```

```
struct InitOk;
```

```
impl RawMutex {  
    fn init(this: PinUninit<'_, Self>) -> InitOk;  
}
```

Abstraction

```
struct PinUninit<'a, T> { ... }

// Unsafe to create token indicating that indeed something is initialised.
struct InitOk;

impl RawMutex {
    fn init(this: PinUninit<'_, Self>) -> InitOk;
}
```

Problem: this is still unsound.

Reentrancy

```
fn rogue_init(this: PinUninit<'_, RawMutex>) -> InitOk {
    static PROOF: Spinlock<Option<InitOk>> = Spinlock::new(None);
    if PROOF.lock().is_some() /* reentrance */ {
        PROOF.lock().take()
    } else {
        let proof = RawMutex::init(this);
        *PROOF.lock() = Some(proof);
        some_func_that_calls_rogue_init();
        loop {}
    }
}
```

Solution: Lifetime branding

```
// Lifetimes of these are made invariant instead of the default covariant.
```

```
struct PinUninit<'a, T> { ... }
```

```
struct InitOk<'a, T> { ... }
```

```
impl<'a, T> PinUninit<'a, T> {
```

```
    unsafe fn init_ok(self) -> InitOk<'a, T>;
```

```
    fn init_with_value(self, value: T) -> InitOk<'a, T>;
```

```
}
```

```
impl RawMutex {
```

```
    fn init<'a>(<
```

```
        this: PinUninit<'a, Self>
```

```
    ) -> InitOk<'a, Self>;
```

```
}
```


Fallible initialisation

```
// Note that branding is still needed for soundness.
```

```
struct InitErr<'a, E> { ... }
```

```
impl<'a, T> PinUninit<'a, T> {
```

```
    fn init_err<E>(self, err: E) -> InitErr<'a, E>;
```

```
}
```

```
impl RawMutex {
```

```
    fn init<'a>(  
        this: PinUninit<'a, Self>
```

```
    ) -> Result<InitOk<'a, Self>, InitErr<'a, Error>>;
```

```
    ) -> Result<InitOk<'a, Self>, InitErr<'a, Error>>;
```

```
}
```

Some transformation

```
type InitResult<'a, T, E> = Result<InitOk<'a, T>, InitErr<'a, E>>;

trait Init<T, E>: Sized {
    fn init<'a>(self, this: PinUninit<'a, T>) -> InitResult<'a, T, E>;
}

fn init_from_closure<T, E, F>(f: F) -> impl Init<T, E>
where
    F: for<'a> FnOnce(PinUninit<'a, T>) -> InitResult<'a, T, E>;

impl RawMutex {
    fn new() -> impl Init<Self, Error>;
}
```

Nice APIs

```
impl<T> PtrPinWith<T> for Box<T> {  
    fn pin_with<E, I>(init: I) -> Result<Pin<Self>, E>  
        where I: Init<T, E>;  
}  
  
// Usage  
let boxed_raw_mutex = Box::pin_with(RawMutex::new()).unwrap();  
  
// Pinning on stack  
init_stack!(raw_mutex_on_stack = RawMutex::new());
```

Structural initialisation

```
struct Mutex<T> {  
    mutex: RawMutex,  
    value: UnsafeCell<T>,  
}  
  
impl<T> Mutex<T> {  
    fn new<F>(value: F) -> impl Init<Self, Error>  
        where F: Init<T, Error>;  
}
```

Structural initialisation

```
struct Mutex<T> {  
    mutex: RawMutex,  
    value: UnsafeCell<T>,  
}
```

```
impl<T> Mutex<T> {  
    fn new<F>(value: F) -> impl Init<Self, Error>  
        where F: Init<T, Error>;  
}
```

Can we write such a new function without any unsafe?

Builder pattern

```
struct MutexBuilder<'this, T>(PinUninit<'this, Mutex<T>>, ...);
```

```
impl<'this, T> MutexBuilder<'this, T> {  
    fn mutex<E, F: Init<RawMutex, E>>(self, f: F)  
        -> Result<Self, InitErr<'this, E>  
  
    fn value<E, F>(self, f: F: Init<T, E>)  
        -> Result<Self, InitErr<'this, E>  
  
    fn finish(self) -> InitOk<'this, Mutex<T>>;  
}
```

```
// Usage
```

```
builder.mutex(RawMutex::new()).value(...).finish()
```

Builder pattern

```
struct MutexBuilder<'this, T>(PinUninit<'this, Mutex<T>>, ...);

impl<'this, T> MutexBuilder<'this, T> {
    fn mutex<E, F: Init<RawMutex, E>>(self, f: F)
        -> Result<Self, InitErr<'this, E>>;

    fn value<E, F>(self, f: F: Init<T, E>)
        -> Result<Self, InitErr<'this, E>>;

    fn finish(self) -> InitOk<'this, Mutex<T>>;
}
```

Builder pattern

```
struct MutexBuilder<'this, T>(PinUninit<'this, Mutex<T>>, ...);

impl<'this, T> MutexBuilder<'this, T> {
    fn mutex<E, F: Init<RawMutex, E>>(self, f: F)
        -> Result<Self, InitErr<'this, E>>;

    fn value<E, F>(self, f: F: Init<T, E>)
        -> Result<Self, InitErr<'this, E>>;

    fn finish(self) -> InitOk<'this, Mutex<T>>;
}
```

How to ensure that each field is initialised once and only once?

Typestates using generics

```
struct MutexBuilder<'this, T, const MUTEX: bool, const VALUE: bool>(...);
```

```
impl<...> MutexBuilder<'this, T, false, VALUE> {  
    fn mutex<E, F: Init<RawMutex, E>>(self, f: F)  
        -> Result<MutexBuilder<'this, T, true, VALUE>, ...>;  
}
```

```
impl<...> MutexBuilder<'this, T, MUTEX, false> {  
    fn value<E, F>(self, f: F: Init<T, E>)  
        -> Result<MutexBuilder<'this, T, MUTEX, true>, ...>;  
}
```

```
impl<'this, T> MutexBuilder<'this, T, true, true> {  
    fn finish(self) -> InitOk<'this, Mutex<T>>;  
}
```

Macro to rescue

```
#[pin_init]
struct Mutex<T> {
    #[pin]
    mutex: RawMutex<T>,
    #[pin]
    value: UnsafeCell<T>,
}

// The macro generates a `MutexBuilder` and
impl<T> Mutex<T> {
    fn builder<'this>(this: PinUinit<'this, Mutex<T>>) -> MutexBuilder<'this, T,
↳ false, false);
}
```

Macro to rescue

```
Box::pin_with(init_pin!(Mutex {  
    mutex: RawMutex::new(),  
    value: UnsafeCell(value)  
}))
```

```
// The `init_pin!` macro expands to  
init_from_closure(move |this| {  
    let builder = Mutex::builder(this);  
    let builder = match builder.mutex(RawMutex::new()) {  
        Ok(v) => v,  
        Err(err) => return Err(err),  
    };  
    ...  
    Ok(builder.finish())  
})
```

Macro to rescue

```
Box::pin_with(init_pin!(Mutex {  
    mutex: RawMutex::new(),  
    value: UnsafeCell(value)  
}))
```

With attribute macro on expressions (unstable feature):

```
Box::pin_with(#[init_pin] Mutex {  
    mutex: RawMutex::new(),  
    value: UnsafeCell(value)  
})
```

What `pin-init` crate include

- ▶ `PinUninit`, `InitOk`, `InitErr` as basic infrastructure
- ▶ Extension traits that add `init_with` and `pin_with` to smart pointers to initialise/create a pinned struct on heap.
- ▶ `init_stack!` to create a pinned struct on stack.
- ▶ `pin_init!` to allow a struct to be initialisable with `init_pin!`
- ▶ Some core types, like `UnsafeCell` and `PhantomPinned`, are made compatible with `init_pin!`.

Drawbacks

- ▶ No way to create self-referential structs safely yet.
- ▶ Needs ability to parse Rust structs and expressions.
- ▶ This method currently depends on `syn`.

Links

- ▶ <https://docs.rs/pin-init>
- ▶ <https://github.com/nbdd0121/pin-init>