pin-init: Solving Address Stability in Rust

Benno Lossin <benno.lossin@proton.me>

15. November 2023

A Motivation for Rust

Initialization should not be that hard...

A Motivation for Rust

Initialization should not be that hard...

```
$ git log --oneline --since 2023-01-01 | grep 'fix.*uninitialized'

cca202a5e595 fbdev: hyperv_fb: fix uninitialized local variable use
fc12a722e6b7 exfat: fix setting uninitialized time to ctime/atime
2a76e7679b59 media: platform: mtk-mdp3: fix uninitialized variable in mdp_path_config()
8f8abb863fa5 net: usb: dm9601: fix uninitialized variable use in dm9601_mdio_read
72151ad0cba8 ASoC: codecs: wsa-macro: fix uninitialized stack variables with name prefix
9147b9ded499 btrfs: fix some -Wmaybe-uninitialized warnings in ioctl.c
e10a35abb3da net: ethernet: mtk_eth_soc: fix uninitialized variable
1c9fd080dffe kunit: fix uninitialized variables bug in attributes filtering
13a0d1088c8f power: supply: qcom_pmi8998_charger: fix uninitialized variable
222a6c42e9ef octeontx2-af: Initialize 'cntr_val' to fix uninitialized symbol error
8362bf82fb54 Input: mcs-touchkey - fix uninitialized use of error in mcs_touchkey_probe()
f72207a5c0db netdevsim: fix uninitialized data in nsim_dev_trap_fa_cookie_write()
f61d2d5cf142 sfc: fix uninitialized variable use
97deb66ed4f9 selftests/mm: fix a "possibly uninitialized" warning in pkey-x86.h
df14afeed2e6 ksmbd: fix uninitialized pointer read in smb2_create_link()
48b47f0caaa8 ksmbd: fix uninitialized pointer read in ksmbd_vfs_rename()
8fd9f4232d81 btrfs: fix an uninitialized variable warning in btrfs_log_inode
0d9b41daa590 nfc: llcp: fix possible use of uninitialized variable in nfc_llcp_send_connect()
714dd3c29a22 phy: mediatek: hdmi: mt8195: fix uninitialized variable usage in pll_calc
8ba7d5f5ba93 btrfs: fix uninitialized variable warnings
c17caf0ba3aa f2fs: fix uninitialized skipped_gc_rwsem
08570b7c8db6 gpu: host1x: fix uninitialized variable use
e88adb4ac27a drm/rockchip: vop2: fix uninitialized variable possible_crtcs
05107edc9101 selftests: sigaltstack: fix -Wuninitialized
7d31677bb7b1 gpu: host1x: fix uninitialized variable use
dc934c183d43 accel/habanalabs: fix a maybe-uninitialized compilation warnings
...
```

A Motivation for Rust

- Rust can help to avoid all of these patches.

A Motivation for Rust

- Rust can help to avoid all of these patches.
- The compiler will statically check for correctness and reject bad code with compile errors.

A Motivation for Rust

- Rust can help to avoid all of these patches.
- The compiler will statically check for correctness and reject bad code with compile errors.
- But an escape hatch is sometimes needed.

A Motivation for Rust

- Rust can help to avoid all of these patches.
- The compiler will statically check for correctness and reject bad code with compile errors.
- But an escape hatch is sometimes needed.
  $\implies$ `unsafe` code is this escape hatch.

A Motivation for Rust

- Rust can help to avoid all of these patches.
- The compiler will statically check for correctness and reject bad code with compile errors.
- But an escape hatch is sometimes needed.
  $\Longrightarrow$ `unsafe` code is this escape hatch.
- But `unsafe` code has its own problems:

A Motivation for Rust

- Rust can help to avoid all of these patches.
- The compiler will statically check for correctness and reject bad code with compile errors.
- But an escape hatch is sometimes needed.
  $\Longrightarrow$ `unsafe` code is this escape hatch.
- But `unsafe` code has its own problems:
  - Similar to C code it's easy to make mistakes.

A Motivation for Rust

- Rust can help to avoid all of these patches.
- The compiler will statically check for correctness and reject bad code with compile errors.
- But an escape hatch is sometimes needed.
  $\Longrightarrow$ `unsafe` code is this escape hatch.
- But `unsafe` code has its own problems:
  - Similar to C code it's easy to make mistakes.
  - Allows unchecked operations that the programmer needs to take care of.

A Motivation for Rust

- Rust can help to avoid all of these patches.
- The compiler will statically check for correctness and reject bad code with compile errors.
- But an escape hatch is sometimes needed.
  $\Longrightarrow$ `unsafe` code is this escape hatch.
- But `unsafe` code has its own problems:
  - Similar to C code it's easy to make mistakes.
  - Allows unchecked operations that the programmer needs to take care of.
  - Needs more careful review.

A Motivation for Rust

- Rust can help to avoid all of these patches.
- The compiler will statically check for correctness and reject bad code with compile errors.
- But an escape hatch is sometimes needed.

  $\Longrightarrow$ `unsafe` code is this escape hatch.
- But `unsafe` code has its own problems:
  - Similar to C code it's easy to make mistakes.
  - Allows unchecked operations that the programmer needs to take care of.
  - Needs more careful review.

  $\Longrightarrow$ try to avoid `unsafe` code.

Address Stability in the Kernel

Why is address stability needed in the Kernel?

Address Stability in the Kernel

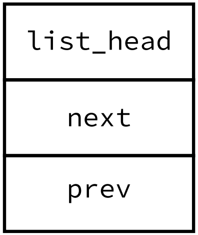
Why is address stability needed in the Kernel?

- Several data structures such as linked lists need to have a stable address, because other structures have pointers to it.

Address Stability in the Kernel

Why is address stability needed in the Kernel?

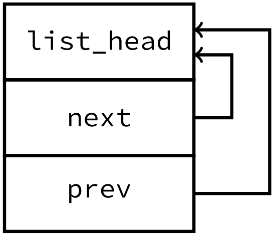
- Several data structures such as linked lists need to have a stable address, because other structures have pointers to it.



Address Stability in the Kernel

Why is address stability needed in the Kernel?

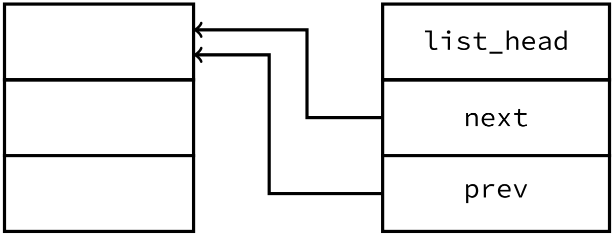
- Several data structures such as linked lists need to have a stable address, because other structures have pointers to it.



Address Stability in the Kernel

Why is address stability needed in the Kernel?

- Several data structures such as linked lists need to have a stable address, because other structures have pointers to it.



Address Stability Support in Rust

- All types are *moveable*.

Address Stability Support in Rust

- All types are *moveable*.
- Moves are common and frequent.

Address Stability Support in Rust

- All types are *moveable*.
- Moves are common and frequent.
- Pointers can be *pinned* by wrapping them in `Pin<P>`

Address Stability Support in Rust

- All types are *moveable*.
- Moves are common and frequent.
- Pointers can be *pinned* by wrapping them in `Pin<P>` for example: `Pin<&mut T>`.

Address Stability Support in Rust

- All types are *moveable*.
- Moves are common and frequent.
- Pointers can be *pinned* by wrapping them in `Pin<P>` for example: `Pin<&mut T>`.
  $\implies$ pointee has a stable address until dropped

Address Stability Support in Rust

- All types are *moveable*.
- Moves are common and frequent.
- Pointers can be *pinned* by wrapping them in `Pin<P>` for example: `Pin<&mut T>`.
  $\Longrightarrow$ pointee has a stable address until dropped
- How does the compiler ensure that no moves happen?

Address Stability Support in Rust

- All types are *moveable*.
- Moves are common and frequent.
- Pointers can be *pinned* by wrapping them in `Pin<P>` for example: `Pin<&mut T>`.
  $\implies$ pointee has a stable address until dropped
- How does the compiler ensure that no moves happen?

```rust
fn swap<T>(a: &mut T, b: &mut T);
```

Address Stability Support in Rust

- All types are *moveable*.
- Moves are common and frequent.
- Pointers can be *pinned* by wrapping them in `Pin<P>` for example: `Pin<&mut T>`.
  $\Longrightarrow$ pointee has a stable address until dropped
- How does the compiler ensure that no moves happen?

```rust
fn swap<T>(a: &mut T, b: &mut T);
```

$\Longrightarrow$ cannot give access to `&mut T` from `Pin<&mut T>`

A Problem with Initialization

A Problem with Initialization

Consider this *bad* piece of C code:

```c
struct list_head new_list_head(void) {
    struct list_head head;
    head.next = &head;
    head.prev = &head;
    return head;
}
```

A Problem with Initialization

Consider this *bad* piece of C code:

```c
struct list_head new_list_head(void) {
    struct list_head head;
    head.next = &head;
    head.prev = &head;
    return head;
}
```

- Rust needs to prevent the equivalent code from compiling.

A Problem with Initialization

Consider this *bad* piece of C code:

```c
struct list_head new_list_head(void) {
    struct list_head head;
    head.next = &head;
    head.prev = &head;
    return head;
}
```

- Rust needs to prevent the equivalent code from compiling.
- To find a solution in Rust, we take a look at the C solution:

A Problem with Initialization

Consider this *bad* piece of C code:

```
struct list_head new_list_head(void) {
    struct list_head head;
    head.next = &head;
    head.prev = &head;
    return head;
}
```

- Rust needs to prevent the equivalent code from compiling.
- To find a solution in Rust, we take a look at the C solution:

```
void init_list_head(struct list_head* head) {
    head->prev = head;
    head->next = head;
}
```

A Problem with Initialization

```c
void init_list_head(struct list_head* head) {
    head->prev = head;
    head->next = head;
}
```

A Problem with Initialization

```
void init_list_head(struct list_head* head) {
    head->prev = head;
    head->next = head;
}
```

Translating to Rust:

A Problem with Initialization

```c
void init_list_head(struct list_head* head) {
    head->prev = head;
    head->next = head;
}
```

Translating to Rust:

```rust
unsafe fn init_list_head(head: *mut ListHead) {
    unsafe {
        (*head).prev = head;
        (*head).next = head;
    }
}
```

A Problem with Initialization

```
void init_list_head(struct list_head* head) {
    head->prev = head;
    head->next = head;
}
```

Translating to Rust:

```
unsafe fn init_list_head(head: *mut ListHead) {
    unsafe {
        (*head).prev = head;
        (*head).next = head;
    }
}
```

But this requires unsafe code!

A Problem with Initialization

```
unsafe fn init_list_head(head: *mut ListHead) {
    unsafe {
        (*head).prev = head;
        (*head).next = head;
    }
}
```

A Problem with Initialization

```
unsafe fn init_list_head(head: *mut ListHead) {
    unsafe {
        (*head).prev = head;
        (*head).next = head;
    }
}
```

Problems with `unsafe`:

- Who ensures that `head` is a valid pointer?

A Problem with Initialization

```
unsafe fn init_list_head(head: *mut ListHead) {
    unsafe {
        (*head).prev = head;
        (*head).next = head;
    }
}
```

Problems with `unsafe`:

- Who ensures that `head` is a valid pointer?
- Still not ensured that `ListHead` stays/even is pinned.

A Problem with Initialization

```
unsafe fn init_list_head(head: *mut ListHead) {
    unsafe {
        (*head).prev = head;
        (*head).next = head;
    }
}
```

Problems with `unsafe`:

- Who ensures that `head` is a valid pointer?
- Still not ensured that `ListHead` stays/even is pinned.
- How to call this without `unsafe`?

A Problem with Initialization

```
unsafe fn init_list_head(head: *mut ListHead) {
    unsafe {
        (*head).prev = head;
        (*head).next = head;
    }
}
```

Problems with `unsafe`:

- Who ensures that `head` is a valid pointer?
- Still not ensured that `ListHead` stays/even is pinned.
- How to call this without `unsafe`?

Solution in C: use convention and rely on the programmer to do it correctly.

A Problem with Initialization

```
unsafe fn init_list_head(head: *mut ListHead) {
    unsafe {
        (*head).prev = head;
        (*head).next = head;
    }
}
```

Problems with `unsafe`:

- Who ensures that `head` is a valid pointer?
- Still not ensured that `ListHead` stays/even is pinned.
- How to call this without `unsafe`?

Solution in C: use convention and rely on the programmer to do it correctly.
Rust aims to offload most of this work to the compiler.

The Solution: pin-init

Turn this:

```
unsafe fn init_list_head(head: *mut ListHead) {
    unsafe {
        (*head).prev = head;
        (*head).next = head;
    }
}
```

The Solution: pin-init

Turn this:

```
unsafe fn init_list_head(head: *mut ListHead) {
    unsafe {
        (*head).prev = head;
        (*head).next = head;
    }
}
```

Into this:

```
fn new() -> impl PinInit<ListHead> {
    pin_init!(&this in ListHead {
        prev: this,
        next: this,
    })
}
```

The Solution: pin-init

```
fn new() -> impl PinInit<ListHead> {
    pin_init!(&this in ListHead {
        prev: this,
        next: this,
    })
}
```

The Solution: `pin-init`

```
fn new() -> impl PinInit<ListHead> {
    pin_init!(&this in ListHead {
        prev: this,
        next: this,
    })
}
```

No `unsafe` to be found!

The Solution: `pin-init`

```
fn new() -> impl PinInit<ListHead> {
    pin_init!(&this in ListHead {
        prev: this,
        next: this,
    })
}
```

No `unsafe` to be found!

The API guarantees:

- All fields of the struct are initialized (none can be forgotten),

The Solution: `pin-init`

```rust
fn new() -> impl PinInit<ListHead> {
    pin_init!(&this in ListHead {
        prev: this,
        next: this,
    })
}
```

No `unsafe` to be found!

The API guarantees:

- All fields of the struct are initialized (none can be forgotten),
- The struct stays pinned after initialization
  (i.e. it will have a stable address),

The Solution: `pin-init`

```
fn new() -> impl PinInit<ListHead> {
    pin_init!(&this in ListHead {
        prev: this,
        next: this,
    })
}
```

No `unsafe` to be found!

The API guarantees:

- All fields of the struct are initialized (none can be forgotten),
- The struct stays pinned after initialization
  (i.e. it will have a stable address),
- No uninitialized memory can be used accidentally,

The Solution: `pin-init`

```
fn new() -> impl PinInit<ListHead> {
    pin_init!(&this in ListHead {
        prev: this,
        next: this,
    })
}
```

No `unsafe` to be found!

The API guarantees:

- All fields of the struct are initialized (none can be forgotten),
- The struct stays pinned after initialization (i.e. it will have a stable address),
- No uninitialized memory can be used accidentally,
- The only way to initialize the struct is `pin-init`,

The Solution: `pin-init`

```rust
fn new() -> impl PinInit<ListHead> {
    pin_init!(&this in ListHead {
        prev: this,
        next: this,
    })
}
```

No `unsafe` to be found!

The API guarantees:

- All fields of the struct are initialized (none can be forgotten),
- The struct stays pinned after initialization (i.e. it will have a stable address),
- No uninitialized memory can be used accidentally,
- The only way to initialize the struct is `pin-init`,
- No runtime cost: it is a zero-cost abstraction.

The Solution: `pin-init`

```
fn new() -> impl PinInit<ListHead> {
    pin_init!(&this in ListHead {
        prev: this,
        next: this,
    })
}
```

No `unsafe` to be found!

The API guarantees:

- All fields of the struct are initialized (none can be forgotten),
- The struct stays pinned after initialization
  (i.e. it will have a stable address),
- No uninitialized memory can be used accidentally,
- The only way to initialize the struct is `pin-init`,
- No runtime cost: it is a zero-cost abstraction.

It is a feature-rich API, so if you need help just ask on zulip:
`https://rust-for-linux.zulipchat.com`

The `pin-init` API in action

Code from the `rust` branch without the `pin-init` API:

The pin-init API in action

Code from the rust branch without the pin-init API:

```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())
```

The pin-init API in action

Code from the rust branch without the pin-init API:

```
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())
```

This requires unsafe code!

The pin-init API in action

Improved code with the pin-init API:

```
pin_init!(Self {
    state_changed <- new_condvar!("SharedState::state_changed"),
    inner <- new_mutex!(
        SharedStateInner { token_count: 0 },
        "SharedState::Inner",
    ),
})
```

No unsafe to be found!

Field Projections

- Having a pointer `&mut Struct` to a struct:

```
struct Struct {
    field: Field,
}
```

Field Projections

- Having a pointer `&mut Struct` to a struct:

```
struct Struct {
    field: Field,
}
```

- Turning that pointer into a pointer "of the same type" to a field of that struct:

Field Projections

- Having a pointer `&mut Struct` to a struct:

  ```
  struct Struct {
      field: Field,
  }
  ```

- Turning that pointer into a pointer “of the same type” to a field of that struct:

  `&mut Struct` ⇝ `&mut Field`

Field Projections

- Having a pointer `&mut Struct` to a struct:

```rust
struct Struct {
    field: Field,
}
```

- Turning that pointer into a pointer "of the same type" to a field of that struct:

$$\texttt{\&mut Struct} \rightsquigarrow \texttt{\&mut Field}$$

  - This is possible in Rust: `&mut my_struct.field`

Field Projections

- Having a pointer `&mut Struct` to a struct:

```
struct Struct {
    field: Field,
}
```

- Turning that pointer into a pointer "of the same type" to a field of that struct:

  $$\texttt{\&mut Struct} \rightsquigarrow \texttt{\&mut Field}$$

  - This is possible in Rust: `&mut my_struct.field`

In Rust a different pointer type can carry additional information:

Field Projections

- Having a pointer `&mut Struct` to a struct:

```
struct Struct {
    field: Field,
}
```

- Turning that pointer into a pointer "of the same type" to a field of that struct:

  $$\texttt{\&mut Struct} \rightsquigarrow \texttt{\&mut Field}$$

  - This is possible in Rust: `&mut my_struct.field`

In Rust a different pointer type can carry additional information:

- `&mut MaybeUninit<Struct>` points to a possibly uninitialized `Struct` (and only provides `unsafe` access)

Field Projections

- Having a pointer `&mut Struct` to a struct:

```rust
struct Struct {
    field: Field,
}
```

- Turning that pointer into a pointer "of the same type" to a field of that struct:

$$\texttt{\&mut Struct} \rightsquigarrow \texttt{\&mut Field}$$

  - This is possible in Rust: `&mut my_struct.field`

In Rust a different pointer type can carry additional information:

- `&mut MaybeUninit<Struct>` points to a possibly uninitialized `Struct` (and only provides `unsafe` access) natural projection:

$$\texttt{\&mut MaybeUninit<Struct>} \rightsquigarrow \texttt{\&mut MaybeUninit<Field>}$$

Field Projections

- Having a pointer `&mut Struct` to a struct:

```
struct Struct {
    field: Field,
}
```

- Turning that pointer into a pointer "of the same type" to a field of that struct:

  `&mut Struct` ⇝ `&mut Field`

  - This is possible in Rust: `&mut my_struct.field`

In Rust a different pointer type can carry additional information:

- `&mut MaybeUninit<Struct>` points to a possibly uninitialized `Struct` (and only provides `unsafe` access) natural projection:

`&mut MaybeUninit<Struct>` ⇝ `&mut MaybeUninit<Field>`

  - But this is not (safely) possible in Rust at the moment

The Problem with `Pin<P>`

- All mutating functions on the pinned type need to take `Pin<&mut Self>`.

The Problem with `Pin<P>`

- All mutating functions on the pinned type need to take `Pin<&mut Self>`.
- Remember: no access to `&mut Self` allowed (because `swap` exists), so how can one modify the fields?

The Problem with `Pin<P>`

- All mutating functions on the pinned type need to take `Pin<&mut Self>`.
- Remember: no access to `&mut Self` allowed (because `swap` exists), so how can one modify the fields?
- For example:

```rust
struct Foo {
    list_head: ListHead,
    count: usize,
}
```

The Problem with `Pin<P>`

- All mutating functions on the pinned type need to take `Pin<&mut Self>`.
- Remember: no access to `&mut Self` allowed (because `swap` exists), so how can one modify the fields?
- For example:

```
struct Foo {
    list_head: ListHead,
    count: usize,
}
```

- `Foo` needs to be pinned because of the `ListHead` field.

The Problem with Pin<P>

- All mutating functions on the pinned type need to take `Pin<&mut Self>`.
- Remember: no access to `&mut Self` allowed (because `swap` exists), so how can one modify the fields?
- For example:

```
struct Foo {
    list_head: ListHead,
    count: usize,
}
```

- `Foo` needs to be pinned because of the `ListHead` field.
- How to modify `count`?

A possible solution: Pin-Projections

```rust
struct Foo {
    list_head: ListHead,
    count: usize,
}
```

A possible solution: Pin-Projections

```rust
struct Foo {
    list_head: ListHead,
    count: usize,
}
```

Observe that there are two types of fields:

A possible solution: Pin-Projections

```
struct Foo {
    list_head: ListHead,
    count: usize,
}
```

Observe that there are two types of fields:

1. Fields that do not require to be pinned (like `count`)

A possible solution: Pin-Projections

```
struct Foo {
    list_head: ListHead,
    count: usize,
}
```

Observe that there are two types of fields:

1. Fields that do not require to be pinned (like `count`)

2. Fields that require to be pinned (like `list_head`)

A possible solution: Pin-Projections

```
struct Foo {
    list_head: ListHead,
    count: usize,
}
```

Observe that there are two types of fields:

1. Fields that do not require to be pinned (like `count`)
   $\implies$ allow access via `&mut usize`
2. Fields that require to be pinned (like `list_head`)

A possible solution: Pin-Projections

```
struct Foo {
    list_head: ListHead,
    count: usize,
}
```

Observe that there are two types of fields:

1. Fields that do not require to be pinned (like `count`)
   $\implies$ allow access via `&mut usize`
2. Fields that require to be pinned (like `list_head`)
   $\implies$ only allow access via `Pin<&mut ListHead>`

A possible solution: Pin-Projections

```
struct Foo {
    list_head: ListHead,
    count: usize,
}
```

Observe that there are two types of fields:

1. Fields that do not require to be pinned (like `count`)
   $\Longrightarrow$ allow access via `&mut usize`
2. Fields that require to be pinned (like `list_head`)
   $\Longrightarrow$ only allow access via `Pin<&mut ListHead>`

- Special case of field projections:

$$\texttt{Pin<\&mut Foo>} \rightsquigarrow \texttt{\&mut usize}$$
$$\texttt{Pin<\&mut Foo>} \rightsquigarrow \texttt{Pin<\&mut ListHead>}$$

A possible solution: Pin-Projections

```rust
struct Foo {
    list_head: ListHead,
    count: usize,
}
```

Observe that there are two types of fields:

1. Fields that do not require to be pinned (like `count`)
   $\implies$ allow access via `&mut usize`
2. Fields that require to be pinned (like `list_head`)
   $\implies$ only allow access via `Pin<&mut ListHead>`

- Special case of field projections:

  `Pin<&mut Foo>` ⇝ `&mut usize`

  `Pin<&mut Foo>` ⇝ `Pin<&mut ListHead>`

- These are called pin projections, they depend on the "intended usecase" of the field and are determined on a field by field basis.

Other Kinds of Projections

The concept of projecting a value is could be useful in many other situations:

Other Kinds of Projections

The concept of projecting a value is could be useful in many other situations:

- `MaybeUninit<T>`: as shown before

Other Kinds of Projections

The concept of projecting a value is could be useful in many other situations:

- `MaybeUninit<T>`: as shown before
- `VolatileMem<T>`: all memory must be accessed by volatile operations

Other Kinds of Projections

The concept of projecting a value is could be useful in many other situations:

- `MaybeUninit<T>`: as shown before
- `VolatileMem<T>`: all memory must be accessed by volatile operations $\implies$ the fields also have to be accessed in a volatile manner. Therefore we can allow projections:

Other Kinds of Projections

The concept of projecting a value is could be useful in many other situations:

- `MaybeUninit<T>`: as shown before
- `VolatileMem<T>`: all memory must be accessed by volatile operations $\Longrightarrow$ the fields also have to be accessed in a volatile manner. Therefore we can allow projections:

  `&mut VolatileMem<Struct>` $\rightsquigarrow$ `&mut VolatileMem<Field>`

Other Kinds of Projections

The concept of projecting a value is could be useful in many other situations:

- `MaybeUninit<T>`: as shown before
- `VolatileMem<T>`: all memory must be accessed by volatile operations $\implies$ the fields also have to be accessed in a volatile manner. Therefore we can allow projections:

  `&mut VolatileMem<Struct>` $\leadsto$ `&mut VolatileMem<Field>`

Another important usage would be for field access via raw pointers:

Other Kinds of Projections

The concept of projecting a value is could be useful in many other situations:

- `MaybeUninit<T>`: as shown before
- `VolatileMem<T>`: all memory must be accessed by volatile operations $\Longrightarrow$ the fields also have to be accessed in a volatile manner. Therefore we can allow projections:

  `&mut VolatileMem<Struct>` $\rightsquigarrow$ `&mut VolatileMem<Field>`

Another important usage would be for field access via raw pointers:

- Allow projecting:

  `*mut Struct` $\rightsquigarrow$ `*mut Field`

Other Kinds of Projections

The concept of projecting a value is could be useful in many other situations:

- `MaybeUninit<T>`: as shown before
- `VolatileMem<T>`: all memory must be accessed by volatile operations $\implies$ the fields also have to be accessed in a volatile manner. Therefore we can allow projections:

  `&mut VolatileMem<Struct>` $\rightsquigarrow$ `&mut VolatileMem<Field>`

Another important usage would be for field access via raw pointers:

- Allow projecting:

  `*mut Struct` $\rightsquigarrow$ `*mut Field`

- Improve ergonomics by introducing projection operator `->`: `foo->bar` instead of `unsafe { addr_of!((*foo).bar) }`

Other Kinds of Projections

The concept of projecting a value is could be useful in many other situations:

- `MaybeUninit<T>`: as shown before
- `VolatileMem<T>`: all memory must be accessed by volatile operations $\Longrightarrow$ the fields also have to be accessed in a volatile manner. Therefore we can allow projections:

  `&mut VolatileMem<Struct>` $\rightsquigarrow$ `&mut VolatileMem<Field>`

Another important usage would be for field access via raw pointers:

- Allow projecting:

  `*mut Struct` $\rightsquigarrow$ `*mut Field`

- Improve ergonomics by introducing projection operator `->`: `foo->bar` instead of `unsafe { addr_of!((*foo).bar) }`
- RFC for adding general field projection support to Rust: `http://github.com/rust-lang/rfcs/pull/3318`

Thanks for Your Attention!

Follow my work:

- RFC for adding general field projection support to Rust: `https://github.com/rust-lang/rfcs/pull/3318`
- `pin-init` userspace library: `https://github.com/Rust-for-Linux/pinned-init`

Contact me on:

- `https://rust-for-linux.zulipchat.com`