Field Projections in the Kernel

Benno Lossin

Pain with `NonNull`

adapted from `rust/kernel/scatterlist.rs:369`:

```
/// # Safety
///
/// `this` must be a valid pointer to `Self`.
unsafe fn project_sgt(this: NonNull<Self>) -> NonNull<RawSGTable> {
    // SAFETY: `this` is a valid pointer to uninitialized memory.
    unsafe { @this.sgt }
}

fn project_sgt_ref(this: &Self) -> &RawSGTable {
    &this.sgt
}
```

Borrow Checker Limitations of `DerefMut`

```
struct RenderState {
    cache: HashMap<String, Arc<String>>,
    template: String,
}

fn render_page(state: &Mutex<RenderState>, name: &str) -> Arc<String> {
    let mut state: MutexGuard<'_, RenderState> = state.lock().unwrap();

    let entry = state.cache.entry(name.to_string());

    entry
        .or_insert_with(|| {
            Arc::new(state.template.replace("$name", name))
        })
        .clone()
}
```

So Many Pointers

- `*const T`, `*mut T`, `NonNull<T>`
- `MyRef<'_, T>`, `MyMut<'_, T>`, `MyBox<T>`
- `Pin<P>`
- `&Cell<T>` `&mut MaybeUninit<T>`
- `Arc<T>` `&UnsafeCell<T>` `Coherent<T>`
- `Rc<T>` `CppRef<T>` `IoMemPtr<T>`
- `Cow<'_, T>` `PyRef<'_, T>` `AtomicPtr<T>`
- `Owned<T>` `MyBox<T>` `&Untrusted<T>`

From Borrow-Checking to Places

```
fn swap_values(value: &mut (i32, i32)) {
    let a = &mut value.0;
    let b = &mut value.1;
    mem::swap(a, b);
}
```

- a *place* is a location in memory, specified through a *place expression*.
  - in this case: the tuples' values, `value.0` and `value.1`.
- *places* are how the borrow checker *analyzes* memory accesses.
  - in this case: the involved places are disjoint:
  - during access analysis of one of them, the borrow checker can ignore the other.
- a *place operation* is what we want to do with a *place*.
  - in this case: borrow mutably.

Place Expressions

Expression Name	Syntax Example
variables & statics	`ptr`, `GLOBAL`
dereferencing	`*ptr`, `**GLOBAL`
field access	`GLOBAL.field`, `(*ptr).field`
indexing	`array[42]`, `(*ptr).field[idx]`
parenthesized place expressions	`(*ptr)`, `((*(*GLOBAL).field).subfield)`

for types implementing `Deref` the deref expression is often implicit: `my_ref.field` for `my_ref: &Struct`

Place Operations

```
struct Value {
    score: i32,
    name: String,
    config: Box<Config>,
}

let value: Value = ...;

// reading:
let _score: i32 = value.score; // i32 is Copy
// moving out:
let name: String = value.name; // String is not Copy

// writing:
value.score = 42;
*value.config = Config::default(); // drops the existing value

// borrowing:
let _: &i32 = &value.score;
let _: *const String = @value.name; // `@`-operator can borrow many pointer types
```

Place Operation Traits

- customizable operations in Rust use a trait: `Add` for `a + b`,

```rust
pub trait PlaceProxy {
    type Target;
}

pub trait ReadPlace: PlaceProxy {
    fn read_place(&self) → Self::Target;
}

pub trait WritePlace: PlaceProxy {
    fn write_place(&mut self, value: Self::Target);
}

pub trait BorrowPlace<X>: PlaceProxy {
    fn borrow_place(&self) → X;
}
```

- how does field access fit into here?
  - what about dereferencing or indexing?
- how do we handle mutable access to one field and shared to another
- → we need an unsafe, low level, borrow-checker escape hatch!

Place Handles

```
pub trait PlaceHandle: Copy {
    type Target;
}
```

- a *place handle* represents the underlying memory location of a pointer
- handles tolerate temporarily invalid borrow-checker states:
  - some fields borrowed or moved out
  - the borrow checker ensures this is corrected before the pointer is accessed again
- handles allow fine-grained access to subplaces
- the type of handles determine and implement the place operations for the pointer they are derived from

```
pub trait CreateHandle: PlaceProxy {
    type Handle: PlaceHandle;

    // ... constructor functions for `Self::Handle`
}
```

- handles are a low-level tool that only the compiler needs to use

From Place Expressions to Handles

Place Expression	Syntax	Handle Equivalent
variables	`var`	`LocalHandle::new(&raw const var)`

```rust
#[lang = "local_handle"] // the compiler can use this type directly
pub struct LocalHandle<T>(*const T);

impl<T> LocalHandle<T> {
    pub unsafe fn new(var: *const T) -> Self {
        Self(var)
    }
}
```

From Place Expressions to Handles

Place Expression	Syntax	Handle Equivalent
field access	`place.field`	`ProjectPlace::project(hdl, field_of!(_, field))`

```rust
pub trait ProjectPlace<S>: PlaceHandle
where
    S: Subplace<Source = Self::Target>,
{
    type Projected: PlaceHandle<Target = S::Target>;

    fn project_place(self, subplace: S) -> Self::Projected;
}
```

From Place Expressions to Handles

Place Expression	Syntax	Handle Equivalent
dereferencing	`*place`	`DerefPlace::deref_place(hdl)`

```
pub trait DerefPlace: PlaceHandle
where
    Self::Target: CreateHandle,
{
    fn deref_place(self) → <Self::Target as CreateHandle>::Handle;
}
```

Operations on Handles

```rust
pub trait PlaceHandle: Copy {
    type Target;
}

pub trait ReadPlace: PlaceHandle {
    fn read_place(self) -> Self::Target;
}

pub trait WritePlace: PlaceHandle {
    fn write_place(self, value: Self::Target);
}

pub trait BorrowPlace<X>: PlaceHandle {
    fn borrow_place(self) -> X;
}
```

Desugaring

```
/// # Safety
///
/// `this` must be a valid pointer to `Self`.
unsafe fn project_sgt(this: NonNull<Self>) -> NonNull<RawSGTable> {
    // SAFETY: `this` is a valid pointer to uninitialized memory.
    unsafe {
        let hdl: LocalHandle<NonNull<Self>> = LocalHandle::new(&raw const this);
        let hdl: NonNullHandle<Self> = DerefPlace::deref_place(hdl);
        let hdl: NonNullHandle<RawSGTable>
            = ProjectPlace::project_place(hdl, field_of!(Self, sgt));
        BorrowPlace::borrow_place(hdl)
    }
}
```

Hidden Place Operations: Dropping

```
fn unbox(b: Box<(String, String)>) → String {
    let first = unsafe {
        let hdl = LocalHandle::new(&raw const b);
        let hdl = DerefPlace::deref_place(hdl);
        let hdl = ProjectPlace::project_place(hdl, field_of!((String, String), 1));
        ReadPlace::read_place(hdl)
    };
    unsafe { // compiler generated
        let hdl = LocalHandle::new(&raw const b);
        let hdl = DerefPlace::deref_place(hdl);
        let hdl = ProjectPlace::project_place(hdl, field_of!((String, String), 1));
        DropPlace::drop_place(hdl);
    };
    unsafe { // compiler generated
        let hdl = LocalHandle::new(&raw const b);
        DropHusk::drop_husk(hdl);
    };
    // now `b.1` and `b` are taken care of and must not be touched again!
    return first;
}
```

Method Resolution

```
struct MyNumGen {
    data: SharedData,
    // ...
}

impl SharedData {
    fn compute_result(self: MyRef<Self>) -> usize { ... }
}

impl Driver for MyNumGen {
    type Ptr = MyRef<Self>;

    fn read(device: Self::Ptr) -> usize {
        SharedData::compute_result(unsafe {
            let hdl = LocalHandle::new(&raw const device);
            let hdl = DerefPlace::deref_place(hdl);
            let hdl = ProjectPlace::project_place(hdl, field_of!(MyNumGen, data));
            BorrowPlace::borrow_place(hdl)
        })
    }
}
```

Match Ergonomics

```
enum MyNumGen {
    Sequential {
        count: usize,
    },
    Random {
        generator: RandomGen,
        // ...
    },
}

let var: IoMemPtr<MyNumGen> = ...;

unsafe match var {
    MyNumGen::Sequential { count } ⇒ {
        let count: IoMemPtr<usize> = count;
        println!("{count}");
    }
    MyNumGen::Random { generator } ⇒ {
        let generator: IoMemPtr<RandomGen> = generator;
        println!("{}", generator.state());
    }
}
```

Place Wrappers

```
1  // from bindgen
2  struct list_head {
3      next: *mut list_head,
4      prev: *mut list_head,
5  }
6
7  fn init_list_head(lh: &mut MaybeUninit<list_head>) {
8      lh.next = MaybeUninit::new(ptr::null_mut());
9      lh.prev = MaybeUninit::new(ptr::null_mut());
10 }
```

- `MaybeUninit<T>` forwards all fields of `T`, by wrapping them in `MaybeUninit`
- also supports arrays!

```
fn init_many_list_heads(lhs: &mut MaybeUninit<[list_head; 64]>) {
    for i in 0..64 {
        let lh: &mut MaybeUninit<list_head> = &mut lhs[i];
        init_list_head(lh);
    }
}
```

Status & Plan

- today:
  - much design work done
  - coherent philosophy for fitting our proposal in with the language
  - writing manual desugaring is possible with https://github.com/BennoLossin/field-projections-designs
  - only a tiny amount of code in `rustc`
- end of this year, we aim for `core` having following new items:
  - operation traits and other items needed for desugaring,
  - the `raw_handle!` macro to perform `unsafe` handle desugaring,
  - `unsafe` macros for the place operations:
    - `borrow!(@place)` (and `borrow!(@<ty> place)`),
    - `read!(place)`, and
    - `write!(place = value)`.
  - macros with no borrow checker support for now (this is why they must be `unsafe`)

Future Work

- borrow checking :)
  - making the place operation macros have borrow checking,
  - *very* difficult question: how do authors of pointer control the borrow checker behavior of their pointer?
- replace the existing semantics of place operations with the new ones!
- what should the safety requirements of the place operation traits be?
- how do we implement this in an efficient and maintainable way in `rustc`?

Use-Cases in the Kernel

Pin

```rust
impl PinnedDrop for GspResources<'_> {
    fn drop(self: Pin<&mut Self>) {
        let device = self.device;
        let bar = self.bar;
        let bundle = self.unload_bundle.take();

        // cleanup ...
    }
}

#[pin_data(PinnedDrop)]
struct GspResources<'gpu> {
    device: &'gpu pci::Device<device::Bound>,
    spec: Spec,
    // ...
    #[pin]
    gsp: Gsp,
    unload_bundle: Option<gsp::UnloadBundle>,
}
```

Volatile Pointers

```
let ptr: IoMemPtr<Struct> = ...;

let x = unsafe { ptr.field }; // uses `volatile_read` under the hood.
unsafe { ptr.field = 42 }; // uses `volatile_write` under the hood.

let _ = unsafe { &ptr.field };
//E               ^^^^^^^^^^ `IoMemPtr` does not allow borrowing as `&`

let _: IoMemPtr<Field> = @ptr.field; // allowed to borrow as `IoMemPtr`
let _: *mut Field = @ptr.field; // allowed to borrow as a raw pointer
```

Tracking Untrusted Data

```
pub struct Untrusted<T>(T);
```

- prevents normal access to the inner value
- only allows access via the `Validate` trait
  - makes all foreign data validation visible & easier to check
  - no accidental usage of unvalidated data
- it is a *place wrapper*:

```
1  let ut: &Untrusted<Struct> = ...;
2  let _: &Untrusted<Field> = &ut.field;
3
4
5  let raw_data: &Untrusted<[u8]> = ...;
6  let _: Untrusted<u8> = raw_data[42];
```

A Safe Abstraction for RCU

```
pub struct Driver { shared: Arc<Mutex<Shared>> }
pub struct Shared { data: Rcu<Box<Data>> }
pub struct Data { num: u32 }

impl Driver {
    pub fn read_data(&self) -> u32 {
        let guard: RcuGuard = read_lock();
        let shared: &Mutex<Shared> = &self.shared;
        let data: &InsideOfMutex<Rcu<Box<Data>>> = &shared.data;
        let data: &Data = data.read(&guard);
        num
    }
}
```

- `Mutex` is a place wrapper that wraps all fields with `InsideOfMutex`
- `InsideOfMutex<T>` doesn't allow any access, except in the case of `T = Rcu<U>`
- `InsideOfMutex<Rcu<U>>` allows creating a reference with the RCU guard held

A Safe Abstraction for RCU

```rust
pub struct Driver { shared: Arc<Mutex<Shared>> }
pub struct Shared { data: Rcu<Box<Data>> }
pub struct Data { num: u32 }

impl Driver {
    pub fn write_data(&self, new_data: Box<Data>) {
        let mut guard: Shield<MutexGuard<'_, Shared>> = self.shared.lock();
        let data: Shield<&mut Rcu<Box<Data>>> = &mut guard.data;
        let old: RcuOld<Box<Data>> = data.write(new_data);
        drop(old); // runs `synchronize_rcu` & drops the old value
    }
}
```

- I partially lied about `write_data`:
- we need the `Shield<P>` pointer wrapper, because writing to an `Rcu<Box< .. >>` must use atomic writes
- but assignment for `&mut Rcu<Box< .. >>` uses non-atomic writes, so `Shield<P>` disallows writes if the pointee of `P` does not implement `Overwrite`
- we do not implement `Overwrite` for `Rcu`
- `Overwrite` must be an auto-trait, so any struct containing a `!Overwrite` field also is `!Overwrite`

Getters & Setters via Field Access

Use-case: exotic layouts, not specifiable in Rust.

- bitfields
- runtime-specified layout
- dynamic size with headers *and* convenient field access

```
let data: &[u8] = ...;
let spec: RuntimeSpec<Struct> = json::load("./spec.json");
let my_struct: RuntimeBox<Struct> = RuntimeBox::copy_deserialize_from(data, spec);

my_struct.score = 42; // uses the offset given in spec
let r: RuntimeRef<'_, Field> = @my_struct.field;
```

- accessing fields calls the place operation trait methods, which read using the (dynamic) offset

Tagged Pointers

```rust
pub struct TaggedPtr<T: TaggedEnum>(*mut T::Rest);

/// # Safety
///
/// - composing `Self::split` and `Self::merge` is the identity.
/// - composing `Self::merge` and `Self::split` is the identity.
pub unsafe trait TaggedEnum {
    type Tag;
    type Rest;

    fn split(self) -> (Self::Tag, Self::Rest);

    unsafe fn merge(tag: Self::Tag, rest: Self::Rest) -> Self;
}

#[derive(TaggedEnum)]
enum MyNumGen {
    Sequential {
        count: usize,
    },
    Random {
        generator: RandomGen,
        // ...
    },
}
```

Discussion

- many more use-cases:
  - virtual pointers
  - permission tracking
  - hashmap insert with index syntax
  - dynamic field names via `my_struct["dynamic_name"]`
- Looking for more kernel use-cases, especially exotic ones