Defining Undefined Behavior in Rust
(Work in Progress)

Ralf Jung, Derek Dreyer

Dagstuhl, May 2018

Max Planck Institute for Software Systems (MPI-SWS)

Rust – Mozilla’s replacement for C/C++

A safe & modern systems PL

Rust – Mozilla’s replacement for C/C++

A safe & modern systems PL

- First-class functions
- Polymorphism/generics
- Traits $\approx$ Type classes incl. associated types

Rust – Mozilla's replacement for C/C++

A safe & modern systems PL

- First-class functions
- Polymorphism/generics
- Traits ≈ Type classes incl. associated types
- Control over resource management (e.g., memory allocation and data layout)

Rust – Mozilla's replacement for C/C++

A **safe** & modern systems PL

- First-class functions
- Polymorphism/generics
- Traits ≈ Type classes incl. associated types
- Control over resource management (e.g., memory allocation and data layout)
- Strong type system guarantees:
    - Type & memory safety; absence of data races

Rust – Mozilla's replacement for C/C++

A

- Firs
- Poly
- Trai
  asso
- Con
  (e.g.
- Stro
  - Type & memory safety; absence of data races

Goal of RustBelt project:
Prove safety of Rust and its standard library.

Rust – Mozilla's replacement for C/C++

Goal of RustBelt project:
Prove safety of Rust and its standard library.
That requires defining what Rust is!

- Firs
- Poly
- Trai asso
- Con (e.g.
- Stro
  - Type & memory safety; absence of data races

Aliasing

+

Mutation

Aliasing

+

Mutation

Rust enforces this via ownership & affine types:

~~Aliasing~~

~~+~~

~~Mutation~~

1. Full ownership: `T`
   + mutation, deallocation
   - aliasing
2. Mutable reference: `&mut T`
   + mutation
   - aliasing, deallocation
3. Shared reference: `&T`
   + aliasing
   - mutation, deallocation

Aliasing guarantees

```
fn answer(x: &mut i32, y: &mut i32) -> i32 {
  // x, y cannot alias: they are unique pointers
  *x = 23;
  *y = 19;
  return *x; // must return 23
}
```

Aliasing guarantees

```rust
fn answer(x: &mut i32, y: &mut i32) -> i32 {
  // x, y cannot alias: they are unique pointers
  *x = 23;
  *y = 19;
  return *x; // must return 23
}

fn shr_read_only(x: &i32) -> bool {
  // x is read-only
  let val = *x;
  unknown_function(x);
  return *x == val; // must return true
}
```

Aliasing guarantees

```
fn answer(x: &mut i32, y: &mut i32) -> i32 {
}

f
  unknown_function(x);
  return *x == val; // must return true
}
```

Rust's reference types provide strong aliasing information.

The optimizer should exploit that.

Aliasing guarantees

```
fn answer(x: &mut i32, y: &mut i32) -> i32 {
}

f
  unknown_function(x);
  return *x == val; // must return true
}
```

Rust's reference types provide strong aliasing information.

The optimizer should exploit that.

But there is a problem: unsafe code.

Unsafe code can access hazardous operations that are banned in safe code:

```
unsafe fn hazardous(x: i32) -> i32 {
  // *const T is the type of raw (unsafe) pointers
  let x_ptr = x as *const i32;
  return *x_ptr;  // dereferencing an integer!
}
```

- Used for better performance, FFI, implementing many standard library types
- Generally encapsulated by safe APIs

```rust
fn mut_dont_alias(x: &mut i32, y: &mut i32) -> i32 {
  // x, y cannot alias: they are unique pointers
  *x = 23;
  *y = 19;
  return *x; // must return 23
}
```

```rust
fn mut_dont_alias(x: &mut i32, y: &mut i32) -> i32 {
  // x, y cannot alias: they are unique pointers
  *x = 23;
  *y = 19;
  return *x; // must return 23
}

fn main() {
  let mut x = 0;
  let x_ptr = &mut x as *mut i32;
  unsafe { // let's pass the same pointer twice
    mut_dont_alias(&mut *x_ptr, &mut *x_ptr);
  }
}
```

Will load 19 because x and y alias!

```
fn mut_dont_alias(x: &mut i32, y: &mut i32) -> i32 {
  // x, y cannot alias: they are unique pointers
  *x = 23;
  *y = 19;
  r
}

fn
  l
  l
  unsafe { // let's pass the same pointer twice
    mut_dont_alias(&mut *x_ptr, &mut *x_ptr);
  }
}
```

Calling `mut_dont_alias` with aliasing pointers must be UB to justify the optimizations.

Quest for a memory model

The memory model for Rust must define how UB arises from unexpected aliasing.

This is work-in-progress. Possible contenders:

1. **Validity-based models** ensure eagerly that variables in scope satisfy aliasing restrictions
2. **Access-based models** enforce aliasing restrictions lazily when references are used

Validity-based memory model

Every memory location gets equipped with a reader-writer lock.

With `x: &mut T`, we hold a write lock.
With `x: &T`, we hold a read lock.

```
fn mut_dont_alias(x: &mut i32, y: &mut i32) -> i32 {
  // x, y cannot alias: they are unique pointers
  *x = 23;
  *y = 19;
  return *x; // must return 23
}

fn main() {
  let mut x = 0;
  let x_ptr = &mut x as *mut i32;
  unsafe { // let's pass the same pointer twice
    mut_dont_alias(&mut *x_ptr, &mut *x_ptr);
  }
}
```

UB because we attempt to acquire a write lock the 2nd time

```
fn split_at_mut(&mut self, mid: usize) ->
  (&mut [T], &mut [T])
{
  let len = self.len();
  // fn as_mut_ptr(&mut self) -> *mut T
  let ptr = self.as_mut_ptr();
  unsafe {
    assert!(mid <= len);
    // fn from_raw_parts_mut<'a, T>(p: *mut T,
    //                     len: usize) -> &'a mut [T]
    (from_raw_parts_mut(ptr, mid),
     from_raw_parts_mut(ptr.offset(mid as isize),
                        len - mid)
    )
  }
}
```

```
fn split_at_mut(&mut self, mid: usize) ->
  (&mut [T], &mut [T])
{
  let len = self.len();
  // fn as_mut_ptr(&mut self) -> *mut T
  let ptr = self.as_mut_ptr();
  unsafe {
    assert!(mid <= len);
    // fn from_raw_parts_mut<'a, T>(p: *mut T,
    //                          len: usize) -> &'a mut [T]
    (from_raw_parts_mut(ptr, mid),
     from_raw_parts_mut(ptr.offset(mid as isize),
                                          id))
```

`self` gets released when calling `as_mut_ptr`

UB because locks get acquired on return value of `from_raw_parts_mut`, overlapping with `self`

Access-based memory model

Keep track of how references are computed, and which reference is *active* for a location.

On every memory access:

- The active reference must be “derived from” the accessing reference.
- The accessing reference becomes the new active reference.

Creating a new reference performs the same check, then makes the new reference active.

```rust
fn mut_dont_alias(x: &mut i32, y: &mut i32) -> i32 {
  // x, y cannot alias: they are unique pointers
  *x = 23;
  *y = 19;
  return *x; // must return 23
}

fn main() {
  let mut x = 0;
  let x_ptr = &mut x as *mut i32;
  unsafe { // let's pass the same pointer twice
    mut_dont_alias(&mut *x_ptr, &mut *x_ptr);
  }
}
```

UB because `x` (last reference to access, and hence active) is not "derived from" `y`.

```
fn split_at_mut(&mut self, mid: usize) ->
  (&mut [T], &mut [T])
{
  let len = self.len();
  // fn as_mut_ptr(&mut self) -> *mut T
  let ptr = self.as_mut_ptr();
  unsafe {
    assert!(mid <= len);
    // fn from_raw_parts_mut<'a, T>(p: *mut T,
    //                              len: usize) -> &'a mut [T]
    (from_raw_parts_mut(ptr, mid),
     from_raw_parts_mut(ptr.offset(mid as isize),
                        len - mid)
    )
  }
}
```

```rust
fn split_at_mut(&mut self, mid: usize) ->
  (&mut [T], &mut [T]
{
  let len = self.len();
  // fn as_mut_ptr(&mut self) -> *mut T
  let ptr = self.as_mut_ptr();
  unsafe {
    assert!(mid <= len);
    // fn from_raw_parts_mut<'a, T>(p: *mut T,
    //                      len: usize) -> &'a mut [T]
    (from_raw_parts_mut(ptr, mid),
     from_raw_parts_mut(ptr.offset(mid as isize),
                                     mid)

}
```

self is not ever used again, hence never becomes active again.

Return value now is active reference for first part, no conflict.

Access-based breaks some optimizations

Suppose `x: &i32` is in scope.

```
loop {
  ...
  let y = *x;
  ...
}
```

$\rightsquigarrow$

```
let tmp = *x;
loop {
  ...
  let y = tmp;
  ...
}
```

Access-based breaks some optimizations

Suppose `x: &i32` is in scope.

```
loop {
  ...
  let y = *x;
  ...
}
```

$\rightsquigarrow$

```
let tmp = *x;
loop {
  ...
  let y = tmp;
  ...
}
```

Problem: What if `*x` is UB, but the code running before it breaks out of the loop?

Memory model trade-offs

Validity-based model:

- Complex to define and reason about
- Program behavior can depend on “lifetime” of reference as inferred by the compiler
- Difficult to make compatible with existing unsafe code

Access-based model:

- Catches errors very late, makes it hard to assign blame
- Requires tracking pointer provenance
- Allows more code, and hence fewer optimizations