Polymorphism in Rust

Traits and Big Lambda

Cooper Pierce & Jack Duvall

Carnegie Mellon University

Attendance

https://forms.gle/5PUkdaptpBgMyXrg6

Outline

Reference Pitfalls

In many other languages with references (e.g., C++) there are a number of potential pitfalls:

```
int main() {
    auto v = std::vector<int>{1, 2, 3, 4};
    auto &x = v[1];
    v.push_back(5);
    x = 0;
    std::cout << v[1] << std::endl;
    return 0;
}
```

What's wrong?

Reference Pitfalls

In many other languages with references (e.g., C++) there are a number of potential pitfalls:

```
int main() {
    auto v = std::vector<int>{1, 2, 3, 4};
    auto &x = v[1];
    v.push_back(5);
    x = 0;
    std::cout << v[1] << std::endl;
    return 0;
}
```

What's wrong?

By changing `v`, we invalidate the reference `x`!

Rules for Borrowing

In Rust, this *cannot happen*, because borrowing has restrictions:

Rules for Borrowing

In Rust, this *cannot happen*, because borrowing has restrictions:

- Every value has an "owner".

Rules for Borrowing

In Rust, this *cannot happen*, because borrowing has restrictions:

- Every value has an “owner”.
- There can only be one owner.

Rules for Borrowing

In Rust, this *cannot happen*, because borrowing has restrictions:

- Every value has an "owner".
- There can only be one owner.
- When ownership of the value ends, the value will be "dropped" (think deallocated/destructed).

Rules for Borrowing

In Rust, this *cannot happen*, because borrowing has restrictions:

- Every value has an “owner”.
- There can only be one owner.
- When ownership of the value ends, the value will be “dropped” (think deallocated/destructed).
- You can have as many shared borrows (`&`) as you want, all at the same time ...

Rules for Borrowing

In Rust, this *cannot happen*, because borrowing has restrictions:

- Every value has an "owner".
- There can only be one owner.
- When ownership of the value ends, the value will be "dropped" (think deallocated/destructed).
- You can have as many shared borrows (`&`) as you want, all at the same time ...
- ... but, you can only have one exclusive borrow (`&mut`), and not at the same time as any shared borrow.

Borrow Rules Prevent Mutable Aliasing

```rust
fn main() {
    let mut v = vec![1, 2, 3, 4];
    let x = &v[0];
    v.push(5);
    println!("{} == {}", x, v[0]);
}
```

Borrow Rules Prevent Mutable Aliasing

```rust
fn main() {
    let mut v = vec![1, 2, 3, 4];
    let x = &v[0];
    v.push(5);
    println!("{} == {}", x, v[0]);
}
```

```
error[E0502]: cannot borrow `v` as mutable because it is also borrowed as immutable
 --> src/main.rs:4:13
  |
3 |             let x = &v[0];
  |                      - immutable borrow occurs here
4 |             v.push(5);
  |             ^^^^^^^^^ mutable borrow occurs here
5 |             println!("{} == {}", x, v[0]);
  |                                  - immutable borrow later used here
```

References: Pointers But Better

References: Pointers But Better

- Reference: “You don’t own this value, but you can still access it”

References: Pointers But Better

- Reference: "You don't own this value, but you can still access it"
    - Value is called "borrowed"

References: Pointers But Better

- Reference: “You don’t own this value, but you can still access it”
  - Value is called ”borrowed”
- Two types: Immutable and Mutable (more accurately: “shared” and “exclusive”)

References: Pointers But Better

- Reference: “You don’t own this value, but you can still access it”
  - Value is called ”borrowed”
- Two types: Immutable and Mutable (more accurately: “shared” and “exclusive”)
- Guarantee: it’s always valid to access memory through a reference!

References: Pointers But Better

- Reference: “You don’t own this value, but you can still access it”
  - Value is called ”borrowed”
- Two types: Immutable and Mutable (more accurately: “shared” and “exclusive”)
- Guarantee: it’s always valid to access memory through a reference!
  - Not the case with pointers

Immutable References

```
&Ty
```

- Only let you read
- Any number can exist at one point, so long as there's no mutable references to the object at the same time.

Immutable References: Example

```
let x: i32 = 5;
let x_ref: &i32 = &x;

// Ok: can have more than one immutable ref
let x_ref2: &i32 = &x;

// Immutable reference is Copy
let x_ref3: &i32 = x_ref;

// Ok: i32 is Copy---can "move out of" reference to one
let y: i32 = *x_ref;
```

Mutable References

```
&mut Ty
```

- Let you read and write
- Can only be made if the underlying object is also mutable
- Only one can exist at a time

Mutable References: Example

```
let x: i32 = 5;

// Error: x isn't mut
let x_mut_ref: &mut i32 = &mut x;

let mut y: i32 = 6;
let y_mut_ref: &mut i32 = &mut y;

// Error: y_mut_ref
let y_mut_ref2: &mut i32 = &mut y;

// Error: mut ref isn't Copy
let y_mut_ref3: &mut i32 = y_mut_ref;
*y_mut_ref += 2;
```

Outline

Generic Code

How would we do it in C?

Generic Code

How would we do it in C?

```c
void *id(void *x) {
    return x;
}
```

Generic Code

How would we do it in C?

```c
void *id(void *x) {
    return x;
}
```

Issues:

Generic Code

How would we do it in C?

```c
void *id(void *x) {
    return x;
}
```

Issues:

- Can only portably use pointers (often violated)

Generic Code

How would we do it in C?

```
void *id(void *x) {
    return x;
}
```

Issues:

- Can only portably use pointers (often violated)
- Normal pointer-related issues in C: null pointers, alignment issues etc...

Generic Code

How would we do it in C?

```
void *id(void *x) {
    return x;
}
```

Issues:

- Can only portably use pointers (often violated)
- Normal pointer-related issues in C: null pointers, alignment issues etc...
- No type-safety

```c
void increment(void *n) {
    *(int*)n += 1;
}
```

```c
void increment(void *n) {
    *(int*)n += 1;
}
```

… but what if I wanted a version for `short`s, `long`s, and so on?

```c
void increment(void *n) {
    *(int*)n += 1;
}
```

… but what if I wanted a version for `short`s, `long`s, and so on?

```c
#define increment(x) _Generic((x),              \
                         short: incr_short, \
                         int: incr_int,     \
                         long: incr_long,   \
                         float: incr_f,     \
                         long double: incr_ld)(x)
```

Another try

```
fun 'a id (x : 'a) : 'a = x
```

Now, properly generic.

Another try

```
fun 'a id (x : 'a) : 'a = x
```

Now, properly generic. (and we only had to switch languages)

Outline

In Rust

```rust
fn id<T>(x: T) -> T {
    x
}
```

In Rust

```
fn id<T>(x: T) -> T {
    x
}
```

In Rust

```rust
fn id<T>(x: T) -> T {
    x
}
```

In Rust

```
fn id<T>(x: T) -> T {
    x
}
```

What about this?

```
fn double<T>(x: T) -> T {
    x + x
}

fn main() {
    println!("{}", double(7));
}
```

Aside: C++ Templates

```
template<typename T>
T id(T x) {
    return x;
}
```

Similar, but not the same.

- Both languages will “monomorphise” this, making a separate version of the function for all of the types it’s used on.
- But in Rust, we typecheck the whole function, not just instances.

Ownership Semantics with Generic Functions

These are still the same as before:

- If the type is Copy, then its copied.
- Otherwise, its moved.

```
fn main() {
    let x = 7;
    let y = String::from("Hello!");
    let z = id(x);
    let w = id(y);
    println!("{}, {}, {}, {}", x, y, z, w);
}
```

Generic Data Structures

So we can be generic over data in our functions, but what about elsewhere?

```
struct Queue<T> {
    in_stack: Vec<T>,
    out_stack: Vec<T>,
}
```

```
enum Option<T> {
    Some(T),
    None,
}
```

Aside: Common Parametric Enums

```
enum Option<T> {
    Some(T),
    None,
}

enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

Lifetime Genericity: Functions

```rust
fn saxpy<'a, 'b>(
    a: f32, x: &'a [f32], y: &'b mut [f32]
) -> &'b mut [f32] {
    for (yi, xi) in y.iter_mut().zip(x) {
        *yi = a * xi + *yi;
    }
    y
}
```

Lifetime Genericity: Data

```rust
enum CopyOnWrite<'a, T> {
    Borrowed(&'a T),
    Owned(T),
}
```

Lifetime Genericity: Data

```
enum CopyOnWrite<'a, T> {
    Borrowed(&'a T),
    Owned(T),
}
```

```
struct Token<'a> {
    range: (usize, usize),
    text: &'a str,
}
```

Lifetime Genericity: Data

```rust
enum CopyOnWrite<'a, T> { Borrowed(&'a T), Owned(T), }

impl<'a, T> CopyOnWrite<'a, T> {
    fn to_mut(&mut self) -> &mut T {
        match self {
            Self::Borrowed(&b) => {
                *self = Self::Owned(b);
                self.to_mut()
            }
            Self::Owned(b) => b,
        }
    }
}
```

Trait Bound Preview

```
enum CopyOnWrite<'a, T> { Borrowed(&'a T), Owned(T), }

impl<'a, T: Copy> CopyOnWrite<'a, T> {
    fn to_mut(&mut self) -> &mut T {
        match self {
            Self::Borrowed(&b) => {
                *self = Self::Owned(b);
                self.to_mut()
            }
            Self::Owned(b) => b,
        }
    }
}
```

Outline

Genericity with Behaviour?

How do we describe a set of behaviours?

Genericity with Behaviour?

How do we describe a set of behaviours?

- Java, C#—interfaces

Genericity with Behaviour?

How do we describe a set of behaviours?

- Java, C#—interfaces
- Plenty of things—abstract classes

Genericity with Behaviour?

How do we describe a set of behaviours?

- Java, C#—interfaces
- Plenty of things—abstract classes
- C++20—concepts

Genericity with Behaviour?

How do we describe a set of behaviours?

- Java, C#—interfaces
- Plenty of things—abstract classes
- C++20—concepts
- Haskell—typeclasses

Genericity with Behaviour?

How do we describe a set of behaviours?

- Java, C#—interfaces
- Plenty of things—abstract classes
- C++20—concepts
- Haskell—typeclasses
- ML—modules

Traits

In Rust, we use a Trait for this.

```
trait PartialEq {
    fn eq(&self, other: &Self) -> bool;
}

trait Bounds {
    fn min() -> Self; // Note the capitalisation!

    fn max() -> Self;
}
```

Traits

In Rust, we use a Trait for this.

```rust
trait PartialEq {
    fn eq(&self, other: &Self) -> bool;
}

trait Bounds {
    fn min() -> Self; // Note the capitalisation!

    fn max() -> Self;
}
```

Traits

In Rust, we use a Trait for this.

```
trait PartialEq {
    fn eq(&self, other: &Self) -> bool;
}

trait Bounds {
    fn min() -> Self; // Note the capitalisation!

    fn max() -> Self;
}
```

Implementing a Trait

Types can then implement traits:

```
impl PartialEq for (i32, i32) {
    fn eq(&self, other: &Self) -> bool {
        self.0 == other.0 && self.1 == other.1
    }
}

impl Bounds for u8 {
    fn min() -> u8 { 0 }
    fn max() -> u8 { 255 }
}
```

Aside: Derive for Implementing Traits

Oftentimes we avoid this for common, boilerplate heavy traits using an "attribute macro"[1].

```
#[derive(Debug, PartialEq, Eq)]
struct Person {
    name: String,
    age: u8,
}
```

Derivable traits include: `Debug`, `PartialEq`, `Eq`, `PartialOrd`, `Ord`, `Clone`, `Copy`, `Hash`, and more.

[1] we'll revisit this in more depth after spring break

Using Trait Implementations

Using a trait implementations is as simple as ensuring the trait is in scope, and just calling the method.

```
trait ToString { fn to_string(&self) -> String; }
impl ToString for i32 { /* omitted */ }

fn main() {
    let s = 7.to_string();
    println!("{}", s);
}
```

Default Implementations

Traits can also include default implementations for their items

```
enum SeekFrom { Start(u64), End(i64), Current(i64), }

/// This trait provides a cursor which can be moved
/// within a stream of bytes.
trait Seek {
    fn seek(&mut self, pos: SeekFrom) -> Result<(), u64>;
    fn rewind(&mut self) -> Result<(), ()> {
        match self.seek(SeekFrom::Start(0)) {
            Ok(_) => Ok(()),
            Err(_) => Err(()),
        }
    }
}
```

Type Parameters for Traits

And much like types, Traits can have type parameters

```
trait From<T> {
    fn from(T) -> Self;
}
```

```
impl From<u8> for i32  { fn from(x: u8)  -> i32 { x as i32 } }
impl From<u16> for i32 { fn from(x: u16) -> i32 { x as i32 } }
impl From<i8> for i32  { fn from(x: i8)  -> i32 { x as i32 } }
impl From<i16> for i32 { fn from(x: i16) -> i32 { x as i32 } }
```

Associated Types

```
trait Iterator {
    type Item;

    fn next(&mut self) -> Option<Self::Item>;

    fn nth(&mut self, n: usize) -> Option<Self::Item> {
        for _ in 0..n {
            self.next()?;
        }
        self.next()
    }
}
```

Note that we can only implement this once for a given type, with some fixed type for `Item`—if many possible types make sense, we should use a type parameter.

Outline

Genericity with Trait Bounds

We can use traits as bounds for our type parameters!

```
fn find_diff<'a, 'b, T: Eq>(
    xs: &'a [T], ys: &'b [T]
) -> Option<(&'a T, &'b T)> {
    for (x, y) in xs.iter().zip(ys) {
        if x != y { return Some((x, y)); }
    }
    None
}
```

Aside: Lifetime Subtyping

```
fn find_same<'a, T: Eq>(xs: &'a [T], ys: &'a [T]) -> Option<&'a T> {
    for (x, y) in xs.iter().zip(ys) {
        if x == y {
            return Some(x);
        }
    }
    None
}
```

Can I use this on any two slices? Do they have to have the exact same lifetime?

Aside: Lifetime Subtyping

```
fn find_same<'a, T: Eq>(xs: &'a [T], ys: &'a [T]) -> Option<&'a T> {
    for (x, y) in xs.iter().zip(ys) {
        if x == y {
            return Some(x);
        }
    }
    None
}
```

Can I use this on any two slices? Do they have to have the exact same lifetime?
No—they can have different ones, and `'a` will be the “shared” lifetime.

Verbose Bounds

Sometimes there can be quite a few constraints, or some complex combination:

```rust
fn double<T>(x: T) -> T
where
    T: Add<T, Output = T> + Copy,
{
    x + x
}
```

Trait Objects: `dyn`

When we use traits in a type parameter bound, we're still monomorphising. What if we want dynamic dispatch?

Trait Objects: `dyn`

When we use traits in a type parameter bound, we're still monomorphising. What if we want dynamic dispatch?

```
trait Button {
    fn on_click(&self, s: State) -> State;
}

fn handle_click_events(
    clicked: &[Box<dyn Button>], mut s: State
) -> State {
    for b in clicked {
        s = b.on_click(s);
    }
    s
}
```

Sized and Unsized Types

Most of the types we've seen so far are "sized", meaning we statically know how large they are.

Some types are "unsized", meaning we don't know their size!

Some examples:

Sized and Unsized Types

Most of the types we've seen so far are "sized", meaning we statically know how large they are.
Some types are "unsized", meaning we don't know their size!
Some examples:

- `[T]`

Sized and Unsized Types

Most of the types we've seen so far are "sized", meaning we statically know how large they are.
Some types are "unsized", meaning we don't know their size!
Some examples:

- `[T]`
- `dyn Trait`

Sized and Unsized Types

Most of the types we've seen so far are "sized", meaning we statically know how large they are.

Some types are "unsized", meaning we don't know their size!

Some examples:

- `[T]`
- `dyn Trait`
- `str` (like `[u8]` but UTF-8)

If we want to use these, they should be through a level of indirection: `&T`, `Box<T>`, etc...

Outline

As a return type

Sometimes we might want to return a specific type which implements a trait, but don't want users of our function to know:

```
enum Tree<T> { Leaf(T), Node(Box<Tree<T>>, T, Box<Tree<T>>) }

struct Leaves { /* omitted */ }
impl Iterator for Leaves { /* omitted */ }

fn leaf_values<T>(tree: &Tree<T>) -> impl Iterator<Item = &T> {
    Leaves { tree, current: tree.leftmost() };
}
```

As a argument's type

This will end up being equivalent to a bound on a type parameter:

```
fn use_fn<T, U>(x: T, f: impl Fn(T) -> U) -> U {
    f(x)
}
```

is the same as

```
fn use_fn<T, U, F: Fn (T) -> U>(x: T, f: F) -> U {
    f(x)
}
```

$$((\exists x. P(x)) \to Q) \iff (\forall x. (P(x) \to Q))$$

Homework

Homework 2 going out likely tomorrow.

- "Due" in a fortnight.
- More complex starter code than last time
- Some later portions touch on next week's content—see writeup
- Ask questions on Discord