Rust programming

Module 2: Foundations of Rust

Unit 4

Traits and Generics

Learning objectives

Traits and generics

The problem

```rust
fn add_u32(l: u32, r: u32) -> u32 { /* -snip- */ }

fn add_i32(l: i32, r: i32) -> i32 { /* -snip- */ }

fn add_f32(l: f32, r: f32) -> f32 { /* -snip- */ }

/* ... */
```

No-one likes repeating themselves

We need generic code!

Generic code

An example

```rust
fn add<T>(lhs: T, rhs: T) -> T { /* - snip - */}
```

Or, in plain English:

- `` `<T>` `` = "let `` `T` `` be a type"
- `` `lhs: T` `` "let `` `lhs` `` be of type `` `T` `` "
- `` `-> T` `` "let `` `T` `` be the return type of this function"

Some open points:

- What can we do with a `` `T` `` ?
- What should the body be?

Bounds on generic code

We need to provide information to the compiler:

- Tell Rust what `` `T` `` can do
- Tell Rust what `` `T` `` is accepted
- Tell Rust how `` `T` `` implements functionality

`trait`

Describe what the type can do

```
trait MyAdd {
    fn my_add(&self, other: &Self) -> Self;
}
```

`impl trait`

Describe how the type does it

```rust
impl MyAdd for u32 {
    fn my_add(&self, other: &Self) -> Self {
      *self + *other
    }
}
```

Using a `trait`

```rust
// Import the trait
use my_mod::MyAdd

fn main() {
  let left: u32 = 6;
  let right: u32 = 8;
  // Call trait method
  let result = left.my_add(&right);
  assert_eq!(result, 14);
  // Explicit call
  let result = MyAdd::my_add(&left, &right);
  assert_eq!(result, 14);
}
```

- Trait needs to be in scope
- Call just like a method
- Or by using the explicit associated function syntax

Trait bounds

```rust
fn add_values<T: MyAdd>(this: &T, other: &T) -> T {
  this.my_add(other)
}

// Or, equivalently

fn add_values<T>(this: &T, other: &T) -> T
  where T: MyAdd
{
  this.my_add(other)
}
```

Now we've got a *useful* generic function!

English: *"For all types* `T` *that implement the* `MyAdd` `trait`*, we define..."*

Limitations of `MyAdd`

What happens if...

- We want to add two values of different types?
- Addition yields a different type?

Making `MyAdd` itself generic

Add an 'Input type' `O`:

```
trait MyAdd<O> {
    fn my_add(&self, other: &O) -> Self;
}

impl MyAdd<u16> for u32 {
    fn my_add(&self, other: &u16) -> Self {
      *self + (*other as u32)
    }
}
```

We can now add a `u16` to a `u32`.

Defining output of `MyAdd`

- Addition of two given types always yields in one specific type of output
- Add *associated type* for addition output

```
trait MyAdd<O> {
    type Output;
    fn my_add(&self, other: &O) -> Self::Output;
}

impl MyAdd<u16> for u32 {
    type Output = u64;

    fn my_add(&self, other: &u16) -> Self::Output {
      *self as u64 + (*other as u64)
    }
}

impl MyAdd<u32> for u32 {
    type Output = u32;

    fn my_add(&self, other: &u32) -> Self::Output {
      *self + *other
    }
}
```

`std::ops::Add`

The way `std` does it

```
pub trait Add<Rhs = Self> {
    type Output;

    fn add(self, rhs: Rhs) -> Self::Output;
}
```

- Default type of `Self` for `Rhs`

`impl std::ops::Add`

```rust
use std::ops::Add;
pub struct BigNumber(u64);

impl Add for BigNumber {
  type Output = Self;

  fn add(self, rhs: Self) -> Self::Output {
      BigNumber(self.0 + rhs.0)
  }
}

fn main() {
  // Call `Add::add`
  let res = BigNumber(1).add(BigNumber(2));
}
```

What’s the type of `res`?

`impl std::ops::Add` (2)

```rust
pub struct BigNumber(u64);

impl std::ops::Add<u32> for BigNumber {
  type Output = u128;

  fn add(self, rhs: u32) -> Self::Output {
      (self.0 as u128) + (rhs as u128)
  }
}

fn main() {
  let res = BigNumber(1) + 3u32;
}
```

What’s the type of `res`?

Traits: Type Parameter vs. Associated Type

Type parameter (input type)

if trait can be implemented for many combinations of types

```rust
// We can add both a u32 value and a u32 reference to a u32
impl Add<u32> for u32 {/* */}
impl Add<&u32> for u32 {/* */}
```

Associated type (output type)

to define a type for a single implementation

```rust
impl Add<u32> for u32 {
  // Addition of two u32's is always u32
  type Output = u32;
}
```

`#[derive]` a `trait`

```rust
#[derive(Clone)]
struct Dolly {
  num_legs: u32,
}

fn main() {
  let dolly = Dolly { num_legs: 4 };
  let second_dolly = dolly.clone();
  assert_eq!(dolly.num_legs, second_dolly.num_legs);
}
```

- Some traits are trivial to implement
- Derive to quickly implement a trait
- For `Clone`: derived `impl` calls `clone` on each field

Orphan rule

*Coherence: There must be **at most one** implementation of a trait for any given type*

Trait can be implemented for a type **iff**:

- Either your crate defines the trait
- Or your crate defines the type

Or both, of course

Compiling generic functions

```rust
impl MyAdd for i32 {/* - snip - */}
impl MyAdd for f32 {/* - snip - */}

fn add_values<T: MyAdd>(left: &T, right: &T) -> T
{
  left.my_add(right)
}

fn main() {
  let sum_one = add_values(&6, &8);
  assert_eq!(sum_one, 14);
  let sum_two = add_values(&6.5, &7.5);
  println!("Sum two: {}", sum_two); // 14
}
```

Code is *monomorphized*:

- Two versions of `add_values` end up in binary
- Optimized separately and very fast to run (static dispatch)
- Slow to compile and larger binary

Common traits from `` `std` ``

Operator overloading: `std::ops::Add<T>` et al.

- Shared behavior

```rust
use std::ops::Add;
pub struct BigNumber(u64);

impl Add for BigNumber {
  type Output = Self;

  fn add(self, rhs: Self) -> Self::Output {
      BigNumber(self.0 + rhs.0)
  }
}

fn main() {
  // Now we can use `+` to add `BigNumber`s!
  let res: BigNumber = BigNumber(1) + (BigNumber(2));
}
```

- Others: `Mul`, `Div`, `Sub`, ...

Markers: `std::marker::Sized`

- Marker traits

```
/// Types with a constant size known at compile time.
/// [...]
pub trait Sized { }
```

`u32` is `Sized`

*Slice `[T]`, `str` is **not** `Sized`*

Slice reference `&[T]`, `&str` is `Sized`

Others:

- `Sync`: Types of which references can be shared between threads
- `Send`: Types that can be transferred across thread boundaries

Default values: `std::default::Default`

```rust
pub trait Default: Sized {
    fn default() -> Self;
}

#[derive(Default)] // Derive the trait
struct MyCounter {
  count: u32,
}

// Or, implement it
impl Default for MyCounter {
  fn default() -> Self {
    MyCounter {
      count: 1, // If you feel so inclined
    }
  }
}
```

Duplication: `std::clone::Clone` & `std::marker::Copy`

```rust
pub trait Clone: Sized {
    fn clone(&self) -> Self;

    fn clone_from(&mut self, source: &Self) {
      *self = source.clone()
    }
}

pub trait Copy: Clone { } // That's it!
```

- Both `Copy` and `Clone` can be `#[derive]`d
- `Copy` is a marker trait
- `trait A: B` == "Implementor of `A` must also implement `B`"
- `clone_from` has default implementation, can be overridden

Conversion: `Into<T>` & `From<T>`

```
pub trait From<T>: Sized {
    fn from(value: T) -> Self;
}

pub trait Into<T>: Sized {
    fn into(self) -> T;
}

impl <T, U> Into<U> for T
  where U: From<T>
{
    fn into(self) -> U {
      U::from(self)
    }
}
```

- Blanket implementation

Prefer `From` over `Into` if orphan rule allows to

Reference conversion: `AsRef<T>` & `AsMut<T>`

```rust
pub trait AsRef<T: ?Sized>
{
    fn as_ref(&self) -> &T;
}

pub trait AsMut<T: ?Sized>
{
    fn as_mut(&mut self) -> &mut T;
}
```

- Provide flexibility to API users
- `T` need not be `Sized`, e.g. slices `[T]` can implement `AsRef<T>`, `AsMut<T>`

Reference conversion: `AsRef<T>` & `AsMut<T>` (2)

```rust
fn print_bytes<T: AsRef<[u8]>>(slice: T) {
  let bytes: &[u8] = slice.as_ref();
  for byte in bytes {
    print!("{:02X}", byte);
  }
  println!();
}

fn main() {
  let owned_bytes: Vec<u8> = vec![0xDE, 0xAD, 0xBE, 0xEF];
  print_bytes(owned_bytes);

  let byte_slice: [u8; 4] = [0xFE, 0xED, 0xC0, 0xDE];
  print_bytes(byte_slice);
}
```

Have user of `print_bytes` *choose between stack local* `[u8; N]` *and heap-allocated* `Vec<u8>`

Destruction: `std::ops::Drop`

```
pub trait Drop {
    fn drop(&mut self);
}
```

- Called when owner goes out of scope

Destruction: `std::ops::Drop`

```rust
struct Inner;

impl Drop for Inner {
  fn drop(&mut self) {
    println!("Dropped inner");
  }
}

struct Outer {
  inner: Inner,
}

impl Drop for Outer {
  fn drop(&mut self) {
    println!("Dropped outer");
  }
}

fn main() {
  // Explicit drop
  std::mem::drop(Outer { inner: Inner });
}
```

Output:

```
Dropped outer
Dropped inner
```

- Destructor runs *before* members are removed from stack
- Signature `&mut` prevents explicitly dropping `self` or its fields in destructor
- Compiler inserts `std::mem::drop` call at end of scope

```rust
// Implementation of `std::mem::drop`
fn drop<T>(_x: T) {}
```

Question: why does `std::mem::drop` work?

Lifetime annotations

What lifetime?

- References refer to variable
- Variable has a lifetime:
  - Start at declaration
  - End at drop

Question: Will this compile?

```
/// Return reference to longest of `&str`s
fn longer(a: &str, b: &str) -> &str {
    if a.len() > b.len() {
        a
    } else {
        b
    }
}
```

```rust
/// Return reference to longest of `&str`s
fn longer(a: &str, b: &str) -> &str {
    if a.len() > b.len() {
        a
    } else {
        b
    }
}
```

```
   Compiling playground v0.0.1 (/playground)
error[E0106]: missing lifetime specifier
 --> src/lib.rs:2:32
  |
2 | fn longer(a: &str, b: &str) -> &str {
  |              ----     ----     ^ expected named lifetime parameter
  |
  = help: this function's return type contains a borrowed value, but the signature does not say whether it is borrowed
help: consider introducing a named lifetime parameter
  |
2 | fn longer<'a>(a: &'a str, b: &'a str) -> &'a str {
  |          ++++     ++          ++          ++

For more information about this error, try `rustc --explain E0106`.
error: could not compile `playground` due to previous error
```

Lifetime annotations

```rust
fn longer<'a>(left: &'a str, right: &'a str) -> &'a str {
    if left.len() > right.len() {
        left
    } else {
        right
    }
}
```

English:

- Given a lifetime called `'a`,
- `longer` takes two references `left` and `right`
- that live for at least `'a`
- and returns a reference that lives for `'a`

Note: Annotations do NOT change the lifetime of variables! Their scopes do!

They just provide information for the borrow checker

Validating boundaries

- Lifetime validation is done within function boundaries
- No information of calling context is used

Question: Why?

Lifetime annotations in types

```
/// A struct that contains a reference to a T
pub struct ContainsRef<'r, T> {
  reference: &'r T
}
```

Lifetime elision

Q: "Why haven’t I come across this before?"

A: "Because of lifetime elision!"

Rust compiler has heuristics for eliding lifetime bounds:

- Each elided lifetime in input position becomes a distinct lifetime parameter.
- If there is exactly one input lifetime position (elided or annotated), that lifetime is assigned to all elided output lifetimes.
- If there are multiple input lifetime positions, but one of them is `` `&self` `` or `` `&mut self` ``, the lifetime of `` `self` `` is assigned to all elided output lifetimes.
- Otherwise, annotations are needed to satisfy compiler

Lifetime elision examples

```
fn print(s: &str);                                     // elided
fn print<'a>(s: &'a str);                              // expanded

fn debug(lvl: usize, s: &str);                         // elided
fn debug<'a>(lvl: usize, s: &'a str);                  // expanded

fn substr(s: &str, until: usize) -> &str;              // elided
fn substr<'a>(s: &'a str, until: usize) -> &'a str;    // expanded

fn get_str() -> &str;                                  // ILLEGAL (why?)

fn frob(s: &str, t: &str) -> &str;                     // ILLEGAL (why?)

fn get_mut(&mut self) -> &mut T;                       // elided
fn get_mut<'a>(&'a mut self) -> &'a mut T;             // expanded
```

Summary