Generics & Traits

CIS 198 Lecture 3

Generics

- Suppose we simplify the `Resultish` enum from last week a bit...

```
enum Result {
    Ok(String),
    Err(String),
}
```

- Better, but it's still limited to passing two values which are both `String`s.

Generics

- This looks a lot like a standard library enum, `Result<T, E>`:

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

- `T` and `E` stand in for any generic type, not only `String`s.
- You can use any CamelCase identifier for generic types.

Generic Structs

- Let's take a look at generic versions of several other structs from last week:

```rust
struct Point<T> {
    x: T,
    y: T,
}

enum List<T> {
    Nil,
    Cons(T, Box<List<T>>),
}
```

Generic Implementations

- To define implementations for structs & enums with generic types, declare the generics at the beginning of the `impl` block:

```
impl<T, E> Result<T, E> {
    fn is_ok(&self) -> bool {
        match *self {
            Ok(_) => true,
            Err(_) => false,
        }
    }
}
```

Traits

- Implementing functions on a per-type basis to pretty-print, compute equality, etc. is fine, but unstructured.
- We currently have no abstract way to reason about what types can do what!

```rust
struct Point {
    x: i32,
    y: i32,
}

impl Point {
    fn format(&self) -> String {
        format!("({}, {})", self.x, self.y)
    }

    fn equals(&self, other: Point) -> bool {
        self.x == other.x && self.y == other.y
    }
}
```

Traits

- Solution: Traits (coming right now)!
- Like we say every week, these are similar to Java interfaces or Haskell typeclasses

Traits

- To define a trait, use a `trait` block, which gives function definitions for the required methods.
  - This is not the same as an `impl` block.
  - Mostly only contains method signatures without definitions.

```
trait PrettyPrint {
    fn format(&self) -> String;
}
```

Traits

- To implement a trait, use an `impl Trait for Type` block.
  - All methods specified by the trait must be implemented.
- One impl block per type per trait.
- You can use `self`/`&self` inside the trait `impl` block as usual.

```rust
struct Point {
    x: i32,
    y: i32,
}

impl PrettyPrint for Point {
    fn format(&self) -> String {
        format!("({}, {})", self.x, self.y)
    }
}
```

Generic Functions

- You can make a function generic over types as well.
- `<T, U>` declares the type parameters for `foo`.
  - `x: T, y: U` uses those type parameters.
- You can read this as "the function `foo`, for all types `T` and `U`, of two arguments: `x` of type `T` and `y` of type `U`."

```rust
fn foo<T, U>(x: T, y: U) {
    // ...
}
```

Generics with Trait Bounds

- Instead of allowing *literally any* type, you can constrain generic types by *trait bounds*.
- This gives more power to generic functions & types.
- Trait bounds can be specified with `T: SomeTrait` or with a `where` clause.
  - "where `T` is `Clone`"

```rust
fn cloning_machine<T: Clone>(t: T) -> (T, T) {
    (t.clone(), t.clone())
}

fn cloning_machine_2<T>(t: T) -> (T, T)
        where T: Clone {
    (t.clone(), t.clone())
}
```

Generics with Trait Bounds

- Multiple trait bounds are specified like `T: Clone + Ord`.
- There's no way (yet) to specify negative trait bounds.
  - e.g. you can't stipulate that a `T` must not be `Clone`.

```
fn clone_and_compare<T: Clone + Ord>(t1: T, t2: T) -> bool {
    t1.clone() > t2.clone()
}
```

Generic Types With Trait Bounds

- You can also define structs with generic types and trait bounds.
- Be sure to declare all of your generic types in the struct header *and* the impl block header.
- Only the impl block header needs to specify trait bounds.
    - This is useful if you want to have multiple impls for a struct each with different trait bounds

Generic Types With Trait Bounds

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

trait PrettyPrint {
   fn format(&self) -> String;
}

impl<T: PrettyPrint, E: PrettyPrint> PrettyPrint for Result<T, E> {
   fn format(&self) -> String {
      match *self {
         Ok(t) => format!("Ok({})", t.format()),
         Err(e) => format!("Err({})", e.format()),
      }
   }
}
```

Examples: Equality

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

// This is not the trait Rust actually uses for equality
trait Equals {
    fn equals(&self, other: &Self) -> bool;
}

impl<T: Equals, E: Equals> Equals for Result<T, E> {
    fn equals(&self, other: &Self) -> bool {
        match (*self, *other) {
            Ok(t1), Ok(t2) => t1.equals(t2),
            Err(e1), Err(e2) => e1.equals(e2),
            _ => false
        }
    }
}
```

- `Self` is a special type which refers to the type of `self`.

Inheritance

- Some traits may require other traits to be implemented first.
  - e.g., `Eq` requires that `PartialEq` be implemented, and `Copy` requires `Clone`.
- Implementing the `Child` trait below requires you to also implement `Parent`.

```
trait Parent {
    fn foo(&self) {
        // ...
    }
}

trait Child: Parent {
    fn bar(&self) {
        self.foo();
        // ...
    }
}
```

Default Methods

- Traits can have default implementations for methods!
  - Useful if you have an idea of how an implementor will commonly define a trait method.
- When a default implementation is provided, the implementor of the trait doesn't need to define that method.
- Define default implementations of trait methods by simply writing the body in the `trait` block.

```
trait PartialEq<Rhs: ?Sized = Self> {
    fn eq(&self, other: &Rhs) -> bool;

    fn ne(&self, other: &Rhs) -> bool {
        !self.eq(other)
    }
}

trait Eq: PartialEq<Self> {}
```

Default Methods

- Implementors of the trait can overwrite default implementations, but make sure you have a good reason to!
  - e.g., *never* define `ne` so that it violates the relationship between `eq` and `ne`.

Deriving

- Many traits are so straightforward that the compiler can often implement them for you.
- A `#[derive(...)]` attribute tells the compiler to insert a default implementation for whatever traits you tell it to.
- This removes the tedium of repeatedly manually implementing traits like `Clone` yourself!

```rust
#[derive(Eq, PartialEq, Debug)]
enum Result<T, E> {
   Ok(T),
   Err(E)
}
```

Deriving

- You can only do this for the following core traits:
  - `Clone`, `Copy`, `Debug`, `Default`, `Eq`,
  - `Hash`, `Ord`, `PartialEq`, `PartialOrd`.
- Deriving custom traits is an unstable feature as of Rust 1.6.
- Careful: deriving a trait won't always work.
  - Can only derive a trait on a data type when all of its members can have derived the trait.
  - e.g., `Eq` can't be derived on a struct containing only `f32`s, since `f32` is not `Eq`.

Core traits

- It's good to be familiar with the core traits.
  - `Clone`, `Copy`
  - `Debug`
  - `Default`
  - `Eq`, `PartialEq`
  - `Hash`
  - `Ord`, `PartialOrd`

Clone

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

    fn clone_from(&mut self, source: &Self) { ... }
}
```

- A trait which defines how to duplicate a value of type `T`.
- This can solve ownership problems.
  - You can clone an object rather than taking ownership or borrowing!

Clone

```
#[derive(Clone)] // without this, Bar cannot derive Clone.
struct Foo {
    x: i32,
}

#[derive(Clone)]
struct Bar {
    x: Foo,
}
```

Copy

```
pub trait Copy: Clone { }
```

- `Copy` denotes that a type has "copy semantics" instead of "move semantics."
- Type must be able to be copied by copying bits (`memcpy`).
  - Types that contain references *cannot* be `Copy`.
- Marker trait: does not implement any methods, but defines behavior instead.
- In general, if a type *can* be `Copy`, it *should* be `Copy`.

Debug

```rust
pub trait Debug {
    fn fmt(&self, &mut Formatter) -> Result;
}
```

- Defines output for the `{:?}` formatting option.
- Generates debug output, not pretty printed.
- Generally speaking, you should always derive this trait.

```rust
#[derive(Debug)]
struct Point {
    x: i32,
    y: i32,
}

let origin = Point { x: 0, y: 0 };
println!("The origin is: {:?}", origin);
// The origin is: Point { x: 0, y: 0 }
```

Default

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

- Defines a default value for a type.

Eq vs. PartialEq

```rust
pub trait PartialEq<Rhs: ?Sized = Self> {
    fn eq(&self, other: &Rhs) -> bool;

    fn ne(&self, other: &Rhs) -> bool { ... }
}

pub trait Eq: PartialEq<Self> {}
```

- Traits for defining equality via the `==` operator.

Eq vs. PartialEq

- `PartialEq` represents a *partial equivalence relation*.
  - Symmetric: if a == b then b == a
  - Transitive: if a == b and b == c then a == c
- `ne` has a default implementation in terms of `eq`.
- `Eq` represents a *total equivalence relation*.
  - Symmetric: if a == b then b == a
  - Transitive: if a == b and b == c then a == c
  - **Reflexive: a == a**
- `Eq` does not define any additional methods.
  - (It is also a Marker trait.)

Hash

```
pub trait Hash {
    fn hash<H: Hasher>(&self, state: &mut H);

    fn hash_slice<H: Hasher>(data: &[Self], state: &mut H)
        where Self: Sized { ... }
}
```

- A hashable type.
- The `H` type parameter is an abstract hash state used to compute the hash.
- If you also implement `Eq`, there is an additional, important property:

  ```
  k1 == k2 -> hash(k1) == hash(k2)
  ```

[1]taken from Rustdocs

Ord vs. PartialOrd

```
pub trait PartialOrd<Rhs: ?Sized = Self>: PartialEq<Rhs> {
    // Ordering is one of Less, Equal, Greater
    fn partial_cmp(&self, other: &Rhs) -> Option<Ordering>;

    fn lt(&self, other: &Rhs) -> bool { ... }
    fn le(&self, other: &Rhs) -> bool { ... }
    fn gt(&self, other: &Rhs) -> bool { ... }
    fn ge(&self, other: &Rhs) -> bool { ... }
}
```

- Traits for values that can be compared for a sort-order.

Ord vs. PartialOrd

- The comparison must satisfy, for all `a`, `b` and `c`:
  - Antisymmetry: if `a < b` then `!(a > b)`, as well as `a > b` implying `!(a < b)`; and
  - Transitivity: `a < b` and `b < c` implies `a < c`. The same must hold for both `==` and `>`.
- `lt`, `le`, `gt`, `ge` have default implementations based on `partial_cmp`.

[1]taken from Rustdocs

Ord vs. PartialOrd

```
pub trait Ord: Eq + PartialOrd<Self> {
    fn cmp(&self, other: &Self) -> Ordering;
}
```

- Trait for types that form a total order.
- An order is a total order if it is (for all `a`, `b` and `c`):
  - total and antisymmetric: exactly one of `a < b`, `a == b` or `a > b` is true; and
  - transitive, `a < b` and `b < c` implies `a < c`. The same must hold for both `==` and `>`.
- When this trait is derived, it produces a lexicographic ordering.

[1]taken from Rustdocs

Associated Types

- Take this `Graph` trait from the Rust book:

```
trait Graph<N, E> {
    fn edges(&self, &N) -> Vec<E>;
    // etc
}
```

- `N` and `E` are generic type parameters, but they don't have any meaningful association to `Graph`
- Also, any function that takes a `Graph` must also be generic over `N` and `E`!

```
fn distance<N, E, G: Graph<N,E>>(graph: &G, start: &N, end: &N)
    -> u32 { /*...*/ }
```

Associated Types

- Solution: associated types!
- `type` definitions inside a trait block indicate associated generic types on the trait.
- An implementor of the trait may specify what the associated types correspond to.

```
trait Graph {
  type N;
  type E;

  fn edges(&self, &Self::N) -> Vec<Self::E>;
}

impl Graph for MyGraph {
  type N = MyNode;
  type E = MyEdge;

  fn edges(&self, n: &MyNode) -> Vec<MyEdge> { /*...*/ }
}
```

Associated Types

- For example, in the standard library, traits like `Iterator` define an `Item` associated type.
- Methods on the trait like `Iterator::next` then return an `Option<Self::Item>`!
    - This lets you easily specify what type a client gets by iterating over your collection.

Trait Scope

- Say our program defines some trait `Foo`.
- It's possible to implement this trait on any type in Rust, including types that you don't own:

```
trait Foo {
   fn bar(&self) -> bool;
}

impl Foo for i32 {
    fn bar(&self) -> bool {
        true
    }
}
```

- But this is really bad practice. Avoid if you can!

Trait Scope

- The scope rules for implementing traits:
  - You need to `use` a trait in order to access its methods on types, even if you have access to the type.
  - In order to write an `impl`, you need to own (i.e. have yourself defined) either the trait or the type.

Display

```
pub trait Display {
    fn fmt(&self, &mut Formatter) -> Result<(), Error>;
}

impl Display for Point {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        write!(f, "Point {}, {})", self.x, self.y)
    }
}
```

- Defines output for the `{}` formatting option.
- Like Debug, but should be pretty printed.
  - No standard output and cannot be derived!
- You can use `write!` macro to implement this without using Formatter.

Addendum: Drop

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

- A trait for types that are destructable (which is all types).
- `Drop` requires one method, `drop`, but you should never call this method yourself.
  - It's inserted automatically by the compiler when necessary.

Addendum: Drop

- Typically, you won't actually implement `Drop` for a type
  - Generally the default implementation is fine.
  - You also don't need to `derive Drop` either.
- Why implement `Drop` then?
  - If you need some special behavior when an object gets destructed.

Addendum: Drop

- Example: Rust's reference-counted pointer type `Rc<T>` has special `Drop` rules:
    - If the number of references to an `Rc` pointer is greater than 1, `drop` decrements the ref count.
    - The `Rc` is actually deleted when the reference count drops to 0.

Addendum: `Sized` vs. `?Sized`

- `Sized` indicates that a type has a constant size known at compile time!
- Its evil twin, `?Sized`, indicates that a type *might* be sized.
- By default, all types are implicitly `Sized`, and `?Sized` undoes this.
  - Types like `[T]` and `str` (no `&`) are `?Sized`.
- For example, `Box<T>` allows `T: ?Sized`.
- You rarely interact with these traits directly, but they show up a lot in trait bounds.

Trait Objects

- Consider the following trait, and its implementors:

```
trait Foo { fn bar(&self); }

impl Foo for String {
    fn bar(&self) { /*...*/ }
}

impl Foo for usize {
    fn bar(&self) { /*...*/  }
}
```

Trait Objects

- We can call either of these versions of `bar` via static dispatch using any type with bounds `T: Foo`.
- When this code is compiled, the compiler will insert calls to specialized versions of `bar`
    - One function is generated for each implementor of the `Foo` trait.

```
fn blah(x: T) where T: Foo {
    x.bar()
}

fn main() {
    let s = "Foo".to_string();
    let u = 12;

    blah(s);
    blah(u);
}
```

Trait Objects

- It is also possible to have Rust perform *dynamic* dispatch through the use of *trait objects*.
- A trait object is something like `Box<Foo>` or `&Foo`
- The data behind the reference/box must implement the trait `Foo`.
- The concrete type underlying the trait is erased; it can't be determined.

Trait Objects

```rust
trait Foo { /*...*/ }

impl Foo for char { /*...*/ }
impl Foo for i32  { /*...*/ }

fn use_foo(f: &Foo) {
    // No way to figure out if we got a `char` or an `i32`
    // or anything else!
    match *f {
        // What type do we have? I dunno...
        // error: mismatched types: expected `Foo`, found `_`
        198 => println!("CIS 198!"),
        'c' => println!("See?"),
        _ => println!("Something else..."),
    }
}

use_foo(&'c'); // These coerce into `&Foo`s
use_foo(&198i32);
```

Trait Objects

- When a trait object is used, method dispatch must be performed at runtime.
  - The compiler can't know the type underlying the trait reference, since it was erased.
- This causes a runtime penalty, but is useful when handling things like dynamically sized types.

Object Safety

- Not all traits can be safely used in trait objects!
- Trying to create a variable of type `&Clone` will cause a compiler error, as `Clone` is not *object safe*.
- A trait is object-safe if:
  - It does not require that `Self: Sized`
  - Its methods must not use `Self`
  - Its methods must not have any type parameters
  - Its methods do not require that `Self: Sized`

[1]taken from Rustdocs

Addendum: Generics With Lifetime Bounds

- Some generics may have lifetime bounds like `T: 'a`.
- Semantically, this reads as "Type `T` must live at least as long as the lifetime `'a`."
- Why is this useful?

Addendum: Generics With Lifetime Bounds

- Imagine you have some collection of type `T`.
- If you iterate over this collection, you should be able to guarantee that everything in it lives as long as the collection.
  - If you couldn't, Rust wouldn't be safe!
- `std::Iterator` structs usually contain these sorts of constraints.