Generic Types, Traits, and Lifetimes

Mustakimur R. Khandaker

Google Online Security Blog: Rust in the Linux kernel

Introduction

Every programming language has tools for effectively handling the duplication of concepts.

- In C++, we have **template**.
- In Rust, one such tool is **generics**.
  - We can express the behavior of generics or how they relate to other generics without knowing what will be in their place.

A **trait** tells the Rust compiler about functionality a particular type has and can share with other types.

- Traits are similar to a feature often called **interfaces** in other languages.

Lifetimes, a variety of generics that give the compiler information about how references relate to each other.

- Lifetimes allow us to borrow values in many situations while still enabling the compiler to check that the references are valid.

Generics

Duplication

```
fn largest_i32(list: &[i32]) -> &i32 {
    let mut largest = &list[0];

    for item in list {
        if item > largest {
            largest = item;
        }
    }

    largest
}
```

```
fn largest_char(list: &[char]) -> &char {
    let mut largest = &list[0];

    for item in list {
        if item > largest {
            largest = item;
        }
    }

    largest
}
```

```
fn code_01() {
    let number_list = vec![34, 50, 25, 100, 65];

    let result = largest_i32(&number_list);
    println!("The largest number is {}", result);

    let char_list = vec!['y', 'm', 'a', 'q'];

    let result = largest_char(&char_list);
    println!("The largest char is {}", result);
}
```

Generics in Function

The function bodies (previous code) have the same code, so let's eliminate the duplication by introducing a generic type parameter in a single function.

```
fn largest<T>(list: &[T]) -> &T {
    let mut largest = &list[0];

    for item in list {
        if item > largest {
            largest = item;
        }
    }

    largest
}
```

```
fn largest<T: PartialOrd>(list: &[T]) -> &T {
    let mut largest = &list[0];

    for item in list {
        if item > largest {
            largest = item;
        }
    }

    largest
}
```

```
error[E0369]: binary operation `>` cannot be applied to type
`&T`
 --> src/main.rs:5:17
  |
5 |         if item > largest {
  |            ---- ^ ------- &T
  |            |
  |            &T
  |
```

Generics in Struct/Enums

We can also define structs/enums to use a generic type parameter in one or more fields using the <> syntax.

- We can implement methods on structs and enums and use generic types in their definitions, too.

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

```
fn code_020() {
    let both_integer = Point { x: 5, y: 10 };
    let both_float = Point { x: 1.0, y: 4.0 };

    println!("p.x = {}", both_integer.x());
    println!("p.x = {}", both_float.x());

    //println!("p.x = {}",
both_integer.distance_from_origin());
    println!("p.x = {}", both_float.distance_from_origin());
}
```

```
impl<T, U> Point<T, U> {
    fn x(&self) -> &T {
        &self.x
    }
}

impl Point<f32, f32> {
    fn distance_from_origin(&self) -> f32 {
        (self.x.powi(2) + self.y.powi(2)).sqrt()
    }
}
```

Note: There is no runtime cost for generics because Rust accomplishes this by performing monomorphization of the code.

Traits

Traits for Structs

Trait definitions are a way to group method signatures together to define a set of behaviors necessary to accomplish some purpose.

```
trait Summary {
    fn summarize(&self) -> String;
    fn details(&self) -> String {
        String::from("(Read more...)")
    }
}
```

```
struct NewsArticle {
    headline: String,
    location: String,
    author: String,
    content: String,
}
```

```
struct Tweet {
    username: String,
    content: String,
    reply: bool,
    retweet: bool,
}
```

```
impl Summary for NewsArticle {
    fn summarize(&self) -> String {
        format!("{}, by {} ({})", self.headline,
self.author, self.location)
    }
}
```

```
impl Summary for Tweet {
    fn summarize(&self) -> String {
        format!("{}: {}", self.username, self.content)
    }
    fn details(&self) -> String {
        format!("{}: {}", self.reply, self.retweet)
    }
}
```

Traits as Parameter

We can use traits to define functions that accept many different types.

```
fn notify(item: &impl Summary) {
    println!("Breaking news! {}", item.summarize());
}
```

```
fn notify(item: &(impl Summary + CustomDisplay)) {
    println!("From Tweeter user {}", item.info());
    println!("Breaking news! {}", item.summarize());
}
```

```
fn returns_summarizable(switch: bool) -> impl Summary
{
    Tweet {
        username: String::from("horse_ebooks"),
        content: String::from("of course ..."),
        reply: false,
        retweet: false,
    }
}
```

```
fn notify<T: Summary>(item: &T) {
    // fn notify<T: Summary>(item1: &T, item2: &T) {
    println!("Breaking news! {}", item.summarize());
}
```

```
    if switch {
        NewsArticle {
            headline: String::from("Penguins ..."),
            location: String::from("Pittsburgh."),
            ...
        }
    } else {
        Tweet {
            username: String::from("horse_ebooks"),
            content: String::from("of course ..."),
            ...
        }
    }
```

Change the largest()

The difference in the following code is that we want to copy the value instead of returning a reference of an array.

```
fn largest<T: PartialOrd>(list: &[T]) -> &T {
    let mut largest = &list[0];

    for item in list {
        if item > largest {
            largest = item;
        }
    }

    largest
}
```

```
fn largest<T: PartialOrd + Copy>(list: &[T]) -> T {
    let mut largest = list[0];

    for &item in list {
        if item > largest {
            largest = item;
        }
    }

    largest
}
```

Lifetimes

Borrow Checker

Most of the time, lifetimes are implicit and inferred, just like most of the time, types are inferred.

- We must annotate lifetimes when the lifetimes of references could be related in a few different ways.

The Rust compiler has a borrow checker that compares scopes to determine whether all borrows are valid.

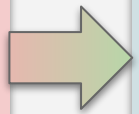
```
{
    let r;                // ---------+-- 'a
                          //          |
    {                     //          |
        let x = 5;        // -+-- 'b  |
        r = &x;           //  |       |
    }                     // -+       |
                          //          |
    println!("r: {}", r); //          |
}                         // ---------+
```

```
{
    let x = 5;            // ----------+-- 'b
                          //           |
    let r = &x;           // --+-- 'a  |
                          //   |       |
    println!("r: {}", r); //   |       |
                          // --+       |
}                         // ----------+
```

Generic Lifetimes in Functions

Let's write a function that returns the longer of two string slices. This function will take two string slices and return a string slice.

```
fn longest(x: &str, y: &str) -> &str
{
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```



```
fn longest<'a>(x: &'a str, y: &'a str) -> &'a str
{
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```

```
error[E0106]: missing lifetime specifier
 --> src/main.rs:9:33
  |
9 | fn longest(x: &str, y: &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
from `x` or `y`
help: consider introducing a named lifetime parameter
  |
9 | fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
  |           ^^^^    ^^^^^^^     ^^^^^^^     ^^^
```

Is that Enough?

```
fn code_07() {
    let string1 = String::from("long string is long");

    {
        let string2 = String::from("xyz");
        let result = longest(string1.as_str(), string2.as_str());
        println!("The longest string is {}", result);
    }
}
```

Will any of the code work?

```
fn code_07() {
    let string1 = String::from("long string is long");
    let result;
    {
        let string2 = String::from("xyz");
        result = longest(string1.as_str(), string2.as_str());
    }
    println!("The longest string is {}", result);
}
```

```
fn longest<'a>(x: &'a str, y: &'a
str) -> &'a str {
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```

Lifetime in Struct

If structs to hold references, in that case we would need to add a lifetime annotation on every reference in the struct's definition.

- Same implementation of the struct.

```
struct ImportantExcerpt<'a> {
    part: &'a str,
}
```

```
impl<'a> ImportantExcerpt<'a> {
    fn level(&self) -> i32 {
        3
    }
    fn announce(&self, announcement: &str) -> &str {
        println!("Attention please: {}",
announcement);
        self.part
    }
}
```

```
fn code_08() {
    let novel = String::from("Call me ...");
    let first_sentence =
novel.split('.').next().expect("Could ...");
    let instance = ImportantExcerpt {
        part: first_sentence,
    };
    //drop(novel);
    passover(instance);
}
```

Implicit Rules

Lifetime elision rules:
Rust programmers found predictable and a few deterministic patterns. So, they programmed these patterns into the compiler's borrow checker to infer implicit lifetimes in these situations.

Definitions:
Lifetimes on function or method parameters are called input lifetimes, and lifetimes on return values are called output lifetimes.

Rules:

- Each parameter that is a reference gets its own lifetime parameter.
- If there is exactly one input lifetime parameter, that lifetime is assigned to all output lifetime parameters.
- If there is atleast one parameter i.e. &self or &mut self, the lifetime of self is assigned to all output lifetime parameters.

```
fn first_word(s: &str) -> &str {
    let bytes = s.as_bytes();

    for (i, &item) in bytes.iter().enumerate() {
        if item == b' ' {
            return &s[0..i];
        }
    }

    &s[..]
}
```

```
fn first_word(s: &str) -> &str {
```

```
fn first_word<'a>(s: &'a str) -> &str {
```

```
fn first_word<'a>(s: &'a str) -> &'a str {
```

```
fn longest(x: &str, y: &str) -> &str {
```

```
fn longest<'a, 'b>(x: &'a str, y: &'b str) -> &str {
```

< Generics, Traits, and Lifetimes />