Please replace the “Trait Bounds” section on page 182-183 with this text, which should then be followed by the existing “Fixing the largest Function with Trait Bounds” section.

Traits as Parameters

Now that you know how to define and implement traits, we can explore how to use traits to define functions that accept many different types.

For example, in Listing 10-13, we implemented the Summary trait on the NewsArticle and Tweet types. We can define a notify function that calls the summarize method on its item parameter, which is of some type that implements the Summary trait. To do this, we can use the impl Trait syntax, like this:

pub fn notify(item: impl Summary) {

println!("Breaking news! {}", item.summarize());

}

Instead of a concrete type for the item parameter, we specify the impl keyword and the trait that the type passed as an argument must implement. In the body of notify, we can call any methods on item that come from the Summary trait, such as summarize. We can call notify and pass in any instance of NewsArticle or Tweet. Code that calls the function with any other type, such as a String or an i32, won’t compile because those types don’t implement Summary.

Trait Bound Syntax

The impl Trait syntax works for straightforward cases but is actually syntax sugar for a longer form, which is called a trait bound; it looks like this:

pub fn notify<T: Summary>(item: T) {

println!("Breaking news! {}", item.summarize());

}

This longer form is equivalent to the example in the previous section but is more verbose. We place trait bounds with the declaration of the generic type parameter after a colon and inside angle brackets.

The impl Trait syntax is convenient and makes for more concise code in simple cases. The trait bound syntax can express more complexity in other cases. For example, we can have two parameters that implement Summary, the impl Trait syntax, using code like this:

pub fn notify(item1: impl Summary, item2: impl Summary) {

If we wanted this function to allow item1 and item2 to have different types, using impl Trait would be appropriate (as long as both types implement Summary). If you wanted to force both parameters to have the same type, that’s only possible to express using a trait bound, like this:

pub fn notify<T: Summary>(item1: T, item2: T) {

The generic type T specified as the type of the item1 and item2 parameters constrains the function such that the concrete type of the value passed as an argument for item1 and item2 must be the same.

Specifying Multiple Trait Bounds with the + Syntax

We can also specify more than one trait bound. Say we wanted notify to use display formatting on item as well as the summarize method: we must specify in the notify definition that item must implement two traits: Display and Summary. We can do so using the + syntax:

pub fn notify(item: impl Summary + Display) {

The + syntax is also valid with trait bounds on generic types:

pub fn notify<T: Summary + Display>(item: T) {

With the two trait bounds specified, the body of notify can call summarize and use {} to format item.

Clearer Trait Bounds with where Clauses

Using too many trait bounds has its downsides. Each generic has its own trait bounds, so functions with multiple generic type parameters can contain lots of trait bound information between the function’s name and its parameter list, making the function signature hard to read. For this reason, Rust has alternate syntax for specifying trait bounds inside a where clause after the function signature. So instead of writing this:

fn some\_function<T: Display + Clone, U: Clone + Debug>(t: T, u: U) -> i32 {

we can use a where clause, like this:

fn some\_function<T, U>(t: T, u: U) -> i32

where T: Display + Clone,

U: Clone + Debug

{

This function’s signature is less cluttered: the function name, parameter list, and return type are close together, similar to a function without lots of trait bounds.

Returning Types that Implement Traits

We can also use the impl Trait syntax in return position to return a value of some type that implements a trait, as shown here:

fn returns\_summarizable() -> impl Summary {

Tweet {

username: String::from("horse\_ebooks"),

content: String::from("of course, as you probably already know, people"),

reply: false,

retweet: false,

}

}

By applying the impl Summary return type, we ensure the returns\_summarizable function returns some type that implements the Summary trait, but it doesn’t specify the concrete type. In this case, returns\_summarizable returns a Tweet, but the code calling this function doesn’t know that.

The ability to return a type that is only specified by the trait it implements is especially useful in the context of closures and iterators, which we cover in Chapter 13. Closures and iterators create types that only the compiler knows or types that are very long to specify. The impl Trait syntax lets you concisely specify that a function returns some type that implements the Iterator trait without needing to write out a very long type.

prod: check xref

However, you can only use impl Trait if you’re returning a single type. For example, this code that returns either a NewsArticle or a Tweet with the return type specified as impl Summary wouldn’t work:

fn returns\_summarizable(switch: bool) -> impl Summary {

if switch {

NewsArticle {

headline: String::from("Penguins win the Stanley Cup Championship!"),

location: String::from("Pittsburgh, PA, USA"),

author: String::from("Iceburgh"),

content: String::from("The Pittsburgh Penguins once again are the best

hockey team in the NHL."),

}

} else {

Tweet {

username: String::from("horse\_ebooks"),

content: String::from("of course, as you probably already know, people"),

reply: false,

retweet: false,

}

}

}

Returning either a NewsArticle or a Tweet isn’t allowed due to restrictions on how you implement impl Trait. We’ll cover how to write a function with this behavior in the section “Using Trait Objects that Allow for Values of Different Types” in Chapter 17 on page XX.

prod: check xref