Types and Traits

Advanced Systems Programming (M) 2018-2019 – Laboratory Exercise 3
Dr Colin Perkins, School of Computing Science, University of Glasgow

1 Introduction

The Advanced Systems Programming (M) course uses the Rust programming language (`https://rust-lang.org/`) to illustrate several advanced topics in systems programming.

In the third lecture, we discussed type-based modelling and design. This is an approach to designing software that involves designing the types first, writing function prototypes using the types to guide the design, then gradually refining and design and implementation. In particular, we discussed the importance of defining specific types that reflect the problem domain, and considered the challenges of implementing type-safe state machines.

This laboratory exercise is intended to give you practice with type driven design. **It is a formative exercise and is not assessed.** You should complete the material at your own pace, asking during one of the scheduled laboratory sessions if you have questions.

2 Numeric Types

A common use of specialist types is to wrap numeric types, to differentiate between different units. The lecture used types representing different units of temperature as an example of this.

To practice wrapping numeric types, implement the standard Rust traits `std::ops::Sub`[1], `std::convert::From`[2], and `std::cmp::Ord`[3] for the `Celsius` and `Fahrenheit` types, along with any other code needed such that the following program compiles and runs:

```
struct Celsius(f32);

struct Fahrenheit(f32);

// Add trait implementations here

fn is_freezing(temp: Fahrenheit) -> bool {
    temp < Celsius(0.0)
}

fn main() {
    let boiling  = Fahrenheit(212.0);
    let freezing = Fahrenheit(32.0);

    let df = boiling - freezing;
    let dc = Celsius::from(df)

    println!("Boiling - freezing is {:?}", diff);   // Fahrenheit(180.0)
    println!("This equals {:?}", dc);               // Celsius(100.0)

    let cold = Fahrenheit(16.0);
    if is_freezing(cold) {
        println!("It's cold!");
    }
}
```

Ask in the laboratory session if you are having difficulty in understanding this code, or the concepts behind using the type system to wrap different numeric types.

[1]`https://doc.rust-lang.org/std/ops/index.html`
[2]`https://doc.rust-lang.org/std/convert/trait.From.html`
[3]`https://doc.rust-lang.org/std/cmp/trait.Ord.html`

3 Traits

Rust uses the concept of *traits* to provide an abstraction for operations that can be performed on different types. Define a trait, `Boiling`, and implement it for the temperature types, along with any other features needed, such that the following program compiles and runs:

```
struct Celsius(f32);

struct Fahrenheit(f32);

// Add trait definitions and implementations here

fn make_tea<T: Boiling>(water_temp: T) {
    if water_temp.is_boiling() {
        println!("Tea's up!");
    } else {
        println!("Water too cold");
    }
}

fn main() {
    let mut water = Celsius(95);

    make_tea(water)
    water += Fahrenheit(15.0);
    make_tea(water)
}
```

Ask in the laboratory session if you are not sure how to do this, or whether you understand concept of traits in Rust.

4 State Machines

It is common to represent network protocols, device drivers, and other behaviours using finite state machines. These state machines can be represented in Rust using either an `enum`-based approach or using a `struct`-based approach.

Read the blog post at `https://insanitybit.github.io/2016/05/30/beyond-memory-safety-with-types`. This outlines how an IMAP client, for retrieving email messages, could be structured using a series of `structs` to represent states and functions to represent state transitions. If you are unclear how the described code work, or what it's implementing, ask in one of the laboratory sessions.

Reflect on the trade-offs between this type of approach to modelling state machines, and the approach using `enums` for states and events as was described in slides 25-27 of lecture 3. In what circumstances do you think the two approaches are most suitable? What are the advantages and disadvantages of the two approaches?

The `struct`-based approach to state machines relies on Rust's ownership rules to ensure state transitions consume the state and release any resources. Ahead of the lecture next week, where we will discuss ownership in detail, read the chapter on ownership in the online Rust book (`https://doc.rust-lang.org/book/ch04-00-understanding-ownership.html`).