Types and Traits

Advanced Systems Programming (H/M) 2024-2025 – Laboratory Exercise 4
School of Computing Science, University of Glasgow

1 Introduction

The Advanced Systems Programming (H) course uses the Rust programming language to illustrate several advanced topics in systems programming. You're expected to learn the basics of programming in Rust as part of this course. This exercise is a further introduction to the Rust programming language, and relates to the content of lectures 3 and 4 of the course. **This is a formative exercise and is not assessed.**

2 Types for Behavioural Abstraction

Rust provides three ways of writing code that is generic across different types. That is, there are three different ways of structuring programs that allow them to work with data without knowing the precise type of that data.

Firstly, **if the data can take one of a known set of types**, it is possible to define an `enum` to describe the possible alternatives, and pattern match on the variants. The use of `enum` values works when the set of possible types a value may hold is known and when there are not many different variants. Importantly, code that uses enumerated types has to either act independently of what variant the data has, or it has to explicitly handle all possible variants. Values are either processed without caring what alternative they encode, or every possible alternative is explicitly listed and considered. Use an `enum` if you know the data can be *this* or *that*, but not any other type.

Secondly, **if the data has a type that is known when it's instantiated, but not when it's defined**, it is possible to write generic data structures and functions that accept type parameters. These make it possible to change the type over which a function operates, or that a data structure holds, at compile time, possibly constrained by type bounds.

Finally, **if the data has unknown type, but that type is known to implement certain methods** then it is possible to define a *trait* to abstract across different types that implement those methods. A `trait` definition specifies the name of the trait, along with a set of prototypes for methods that all instances of the trait must implement, but it provides no bodies for those methods. For example, a trait `Area` that defines a single method, `area()`, can be expressed as follows:

```
trait Area {
    fn area(&self) -> u32;
}
```

This is similar to an interface definition in a language such as Java. It indicates that types that implement the trait *must* provide implementations of the specified functions. Such implementations are provided by an `impl` block specifying the trait name and type for which it's implemented. For example, the `Area` trait mentioned above could be implemented for a type, `Rectangle`, as follows:

```
impl Area for Rectangle {
    fn area(&self) -> u32 {
        self.width * self.height
    }
}
```

This would make the method, `area()`, available on instances of the `Rectangle` type. The `Rectangle` type will often be a structure type, but other types can implement traits too.

Traits are often used to solve the same types of problems that subclasses solve in object-oriented languages such as Java, or that duck-typing solves in dynamic languages such as Python. They make it possible to implement code that works with any type that implements a particular set of methods. This is done by specifying a *trait bound* as part of the type parameter in a structure or function definition. For example, the code below defines a trait, `Summary`, and a function, `notify()`. The definition of the `notify()` function takes a type parameter, `T`, where the definition of the type parameter (given in angle brackets) indicates that the actual type used when the function is invoked *must* implement the trait `Summary`. That is, the `notify()` function can be called with any `item`, provided that `item` is of a type that implements the `Summary` trait.

```
trait Summary {
    fn summarize(&self) -> String;
}

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

Trait bounds can also be specified in type parameters in other places, including structure definitions, and constrain the types that can be used to instantiate the parameter.

It's also possible to define a function or method that returns some value where the exact type of the return value is not known, but it's known to implement some trait. This is written using `impl T` in the return type, where `T` is a trait name or type parameter with a trait bound. For example, if a `Document` trait were defined, a function to correct the spelling in any type of document could be written:

```
fn spell_check<D: Document>(document: D) -> impl D {
    // .. implementation omitted
}
```

The use of trait bounds looks, at first glance, to be similar to the use of classes in object-oriented languages. The meaning is different, though. In an object-oriented language, a bound such as that above indicates that the `spell_check()` function can take anything that *is a* `Document` as a parameter; in Rust, it indicates that it takes anything that *behaves like* a `Document`.

Read Sections 10.1 and 10.2 of the Rust book (`https://doc.rust-lang.org/book/ch10-00-generics.html`) and work through the examples. The final part of the chapter, on "Validating References with Lifetimes", relates to the Rust ownership and borrowing rules that we'll discuss in Lecture 5, and shouldn't be attempted at this time.

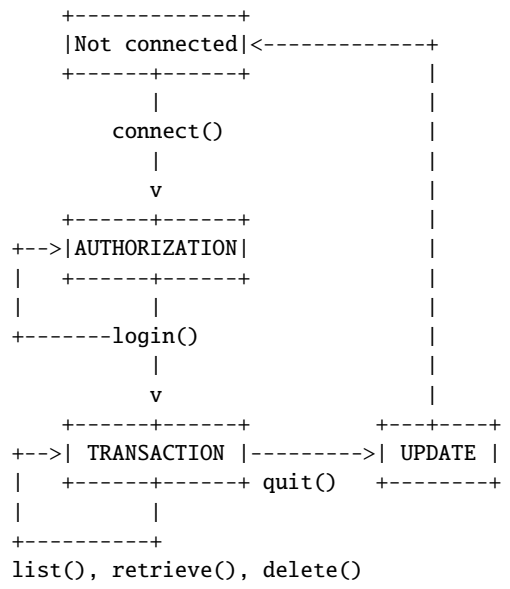
3 Formative Exercise

One of the benefits of using a programming languages with an expressive type system is that it becomes possible to develop an effective model of the problem domain using the type system, and to have the compiler help you validate that model for consistency and correctness.

At its most basic, this can allow the programmer to cleanly represent the different types of data present in the system, along with set of the legal operations that can be performed on those objects, and have the compiler check their use for consistency. Rather than return an `int`, a program might return a `FileDescriptor`; rather than a `char *`, it might return a `Username`; and so on. Many programming languages have type systems that support this level of expressivity. If applied systematically, use of such types can effectively catch many small bugs.

More expressive languages can also encode state machines and behaviours, allowing the compiler to help check more complex systems for consistency. This introduces some up-front design and modelling costs, but reduces later debugging effort.

For example, consider the POP3 email retrieval protocol, described by the IETF in RFC1939.[1] This is a simple, and now rather obsolete, email retrieval protocol. An email client, for example a previous version of Microsoft Outlook, connects to an email server. After connecting, it logs in. And after successfully logging in, it can list, retrieve, and delete email messages from the inbox. The state machine for the protocol is structured as follows:



In more detail, the behaviour in the different states is:

- When not connected to the email server, the only possible operation is to connect to the server, putting the system into the AUTHORIZATION state.
- Once connected in the AUTHORIZATION state, the user can login by sending a username and password. If these are correct, the system enters the TRANSACTION state; otherwise it returns to AUTHORIZATION.
- In the TRANSACTION state, the user can list the available email messages, returning a list of message identifiers; they can retrieve or delete messages based on their identifiers, or they can quit.
- Quitting moves the system into the UPDATE state, where any changes made are saved and a status code returned, before the system disconnects from the email server.

Read RFC 1939 and study the above state-transition diagram, to get a high-level understanding of the behaviour of the POP3 protocol and to practice reading network protocol specifications. Sketch out a design for a POP3 server implementation, showing only the types, methods, traits, and function prototypes, and omitting the body of any methods or functions. This should use Rust `structs` to represent the various states, with methods on those structures representing operations and state transitions. Think about how the Rust ownership rules, which we started to discuss and will consider in more detail in Lecture 5, help to enforce that operations are performed only in the correct order and in the correct state. Consider what types might be needed, in addition to those representing the states.

This is a formative exercise and is not assessed. You do not need to submit your solution. An outline solution will be discussed as part of Lecture 5, and you should also discuss your solution with the lecturer or lab demonstrator.

[1] `https://datatracker.ietf.org/doc/html/rfc1939`