Homework 5: Type State

Due date: November 10 (Wednesday) at 11:59pm

Overview

This homework consists of two parts:

1. Using type state, implement a shopping cart API that follows a given state machine specification.
2. Using type state, implement a network client API that follows a given state machine specification. Then, using the client API, implement a function that sends a set of packets to an unreliable server that may drop some packets.

Important notes:

- You will use the Rust language, in particular **Rust 1.47.0**. You may either install it on your local machine or work on the `myth` cluster. For the first option, install the most recent Rust by following the instruction on `https://rustup.rs/`, and switch to Rust 1.47.0 by entering the following commands.

  ```
  rustup install 1.47.0
  rustup default 1.47.0
  ```

  After the installation, The command `rustc --version` should output `rustc 1.47.0 (...)`.

- Read the following guide to Rust before starting the homework.

  `https://cs242.stanford.edu/materials/assignments/rust-guide.html`

  For more details on Rust, see `https://doc.rust-lang.org/book/`.

- You are optionally allowed to work with a partner for this assignment. One member of each pair should submit the assignment on Canvas (instructions at the bottom).

Type State in Rust

The state machine diagram in Figure 1 can be implemented in Rust as in Figure 2. In the implementation, each state (e.g., StateA) is translated to a distinct struct (e.g., `StateA`), and each struct contains only the methods (e.g., `inc`) accessible in the corresponding state. Moreover, the return type of each method (e.g., `inc`) is the struct type (e.g., `StateB`) that corresponds to the appropriate state transition in the diagram. Note that a result type (i.e., `Result<StateA,()>`) is used in `StateB` to handle the success and failure cases separately. Finally, the first argument of `inc` and `half` is `self`, not `&self`.

All of these features of the implementation in Figure 2 ensure that invalid uses of the methods and objects (which violate the state machine diagram in Figure 1) are rejected by the Rust type system. For example, the following two code segments raise compile errors: `let a = Foo::new(); let b = a.half();` and `let a = Foo::new(); let b1 = a.inc(1); let b2 = a.inc(2);`. The first code segment is invalid because the method `half` is called in the StateA state, and the second one is invalid because the object `a` is used after being used to create `b1`. On the other hand, valid uses of the methods and objects are verified by the type system. For instance, the following code segment compiles and runs correctly: `let a = Foo::new(); let b = a.inc(2); let ok_a = b.half();`. In this homework, you will apply the type state design pattern illustrated above to implement two APIs that have state machine specifications.

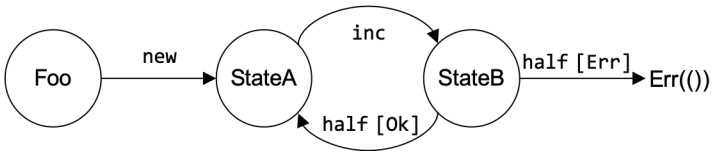

Figure 1: Example state machine.

```
pub struct Foo    { }
pub struct StateA { n: u32 }
pub struct StateB { n: u32 }
impl Foo {
    pub fn new() -> StateA {
        StateA { n: 0 }
    }
}
impl StateA {
    pub fn inc(self, i) -> StateB {
        StateB { n: self.n + i }
    }
}
```

```
impl StateB {
    pub fn half(self) -> Result<StateA,()> {
        if (self.n % 2 == 0) {
            Ok(StateA { n: self.n / 2 })
        } else {
            Err(())
        }
    }
}
```

Figure 2: Rust implementation of the example state machine.

Part 1: Shopping Cart

Problem 1: Cart API

The goal of this problem is to implement an API for a simple shopping cart application that should conform to the state machine diagram in Figure 3. The diagram shows a list of methods accessible in each state, and the state transitions caused by each method call. For example, in the Cart state, you can access the `login` method only. If the login succeeds, control moves to the Empty state which can only access the `add_item` method. If the login fails, the API returns `Err(())` (see below for details).

The method `login` tries to login with a given user id and password. The method `add_item` adds the cost of an item to the cart. The method `clear_items` removes all the items from the cart. The method `checkout` freezes the cart—no more items can be added. The method `cancel` undoes the freeze action taken by `checkout`. The method `order` removes all items from the cart as `clear_items` does.

In addition to conforming to the diagram, the cart API should also have correct types for arguments and return values so that it can compile and run the provided test cases (e.g., Figure 4). Below we list some details of the method signatures of the cart API. For more details, see `part1/src/prob1/cart.rs`.

- The only two methods in the cart API that receive one or more arguments are `login` and `add_item`: `login` takes a user id and password as two `String`s, and `add_item` takes a cost of an item as an `u32`.
- The method `login` returns `ret` of type `Result<T,()>` for some type `T`, where `()` denotes the unit type and `T` will depend on your implementation. Note that for any types `A` and `B`, the result type `Result<A,B>` is a built-in type in Rust that can have two kinds of objects: `Ok(a)` for any `a` of type `A`, and `Err(b)` for any `b` of type `B`. The method `login` returns `ret=Ok(t)` for some `t` of type `T` if the login succeeds, where `t` denotes a cart object in the Empty state. If the login fails, `login` returns `ret=Err(())`.
- The two methods in the cart API not shown in the state diagram are `acct_num` and `tot_cost`. Both methods are accessible in all states except the Cart state, and return an `u32`. See below for details.

Use type state to implement the cart API in `part1/src/prob1/cart.rs` so that every valid API usage should compile and run correctly, and every invalid API usage should not compile. In your implementation, each state should correspond to the struct **whose name is the same as the state**

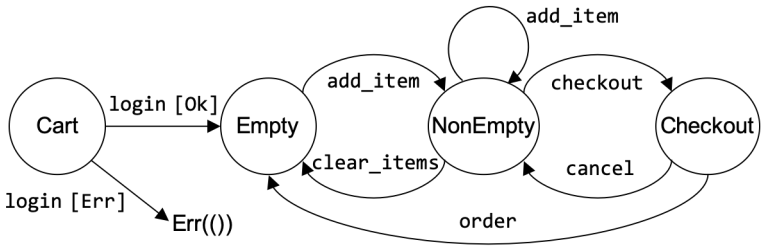

Figure 3: State machine diagram for the cart API.

```
match Cart::login("id1".to_string(), "pw1".to_string()) {
    Err(()) =>
        println!("Login failed!"),
    Ok(empty) => {
        let nonempty = empty.add_item(1);
        let nonempty = nonempty.add_item(2);
        let checkout = nonempty.checkout();
        let empty    = checkout.order();
        println!("acct_num = {}", empty.acct_num());
        println!("tot_cost = {}", empty.tot_cost());
    }
}
```

Figure 4: A segment of code that uses the cart API.

name (e.g., the Cart state will correspond to the struct `Cart`). The signatures of the methods to implement are provided in the starter code; make sure to use them exactly. The reference solution without comments is 52 lines. Detailed instructions:

- In the `login` method, you should call `internal_login(...)` provided in `part1/src/prob1/server.rs` to check if login succeeds.
- The `acct_num` method should return the account number that the `login` method obtained when it called `internal_login(...)`. The `tot_cost` method should return the total cost of all the items in the cart.
- To test your implementation, change the current directory to `part1/`, and run `cargo test prob1::valid -- --show-output` and `cargo test prob1::invalid -- --show-output`. The command with `prob1::valid` checks that valid API usage does compile and runs correctly, and the command with `prob1::invalid` checks that invalid API usage does not compile.
  - For each test command, the last (or second last) line of its output tells you how many test cases you fail. If you see `test result: ok. ... 0 failed; ...`, that means your solution passes all the given test cases.
  - Make sure that the files `part1/wip/cart_*.stderr` generated by the invalid test cases are identical to the files `part1/wip/expected_cart_*.stderr` which we provided, aside from line & column numbers that depend on your implementation (i.e. in `$DIR/cart.rs:[line]:[col]`, line and col can be anything).
  - You may encounter the following warning: "warning: Hard linking files in the incremental compilation cache failed." If so, please ignore them.
- The test cases are in `part1/tests/prob1/mod.rs`. Note that the test cases are incomplete—it is your task to ensure that valid API usages should compile and invalid ones should not.

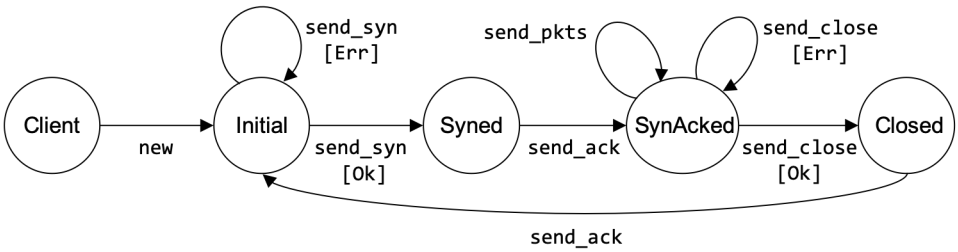

Figure 5: State machine diagram for the client API.

Part 2: Network Client

In the second part of this assignment, you will implement a network client API that should follow a state machine specification (Problem 2), and implement a function, using the client API, that should send a set of packets to an unreliable server that may drop some packets (Problem 3).

Problem 2: Client API

The goal of this problem is to implement an API for a simple network client application that should conform to the state machine diagram in Figure 5 (a simplified version of the TCP protocol). As in Problem 1, the diagram shows the methods allowed in each state and the state transition policy.

Each method, except `new`, shown in the diagram sends a signal or a vector of packets to a server, and receives some messages from the server as a response. A signal can be one of the following: `Syn`, `Ack`, `Close`, `SynAck`, and `CloseAck`. A client can send one of the first three signals to a server, and as a response to the send, the server can send back to the client one of the last two signals. A packet consists of two parts: a buffer (`buf`) of type `char` and an id (`id`) of type `u32`. The `buf` part denotes the actual data of the packet, and the `id` part denotes the id of the packet. You can assume all packets have distinct ids. The types of a signal and a packet, denoted by `Sig` and `Pkt`, are defined in `part2/src/prob2/msg.rs`.

Each of the methods `send_syn`, `send_ack`, and `send_close` sends to a server each of the signals `Syn`, `Ack`, and `Close`, respectively. For the signals `Syn` and `Close`, a server may fail to receive them, and in such a case the client receives no signal back from the server. If a server receives `Syn` or `Close` successfully, the client receives `SynAck` or `CloseAck`, respectively, back from the server (i.e., the server acknowledges that it received `Syn` or `Close`). For the signal `Ack`, a server always receives it successfully, but a server does not send any signal back to the client when it receives `Ack`.

The Initial state moves to the Syned state iff `send_syn` is called and the client receives `SynAck` from the server in response to sending `Syn`. Similarly, the SynAcked state moves to the Closed state iff `send_close` is called and the client receives `CloseAck` from the server in response to sending `Close`. Finally, the Syned and Closed states move to the next states after `send_ack` is called and the client sends `Ack` to the server.

The method `send_pkts` tries to send given packets `pkts` to a server. As in the case for signals, a server may fail to receive some parts of `pkts`. Fortunately, the server sends back to the client the ids of the packets that the server received successfully. For instance, a client could send the packets `vec![Pkt{buf:'a',id:1}, Pkt{buf:'b',id:2}, Pkt{buf:'c',id:3}]` to a server, and the server could fail to receive the first packet only; in that case, the server sends back to the client the ids `vec![2,3]`.

In addition to conforming to the diagram, the client API should also have correct types for arguments and return values so that it can compile and run the provided test cases (e.g., Figure 6). Below we list some details of the method signatures of the client API. For more details, see `part2/src/prob2/client.rs`.

- Every method except `new` and `ids_sent` takes a server (as a `&mut Server`) as the first argument. The only method in the client API that receives additional arguments is `send_pkts`. The method takes a vector of packets (as a `&Vec<Pkt>`) to be sent to a server as the second argument.
- The methods `send_syn` and `send_close` return a value of types `Result<T1,T2>` and `Result<T3,T4>`, respectively, for some types `Ti` which will depend on your implementation.

```
let mut server: Server = Server::new(...);
let pkts: Vec<Pkt> = vec![...];
let initial = Client::new();
match initial.send_syn(&mut server) {
    Err(_syn_failed) =>
        println!("Syn failed!"),
    Ok(syned) => {
        let synacked = syned.send_ack(&mut server);
        let synacked = synacked.send_pkts(&mut server, &pkts);
        match synacked.send_close(&mut server) {
            Err(_close_failed) =>
                println!("Close failed!"),
            Ok(closed) => {
                println!("Packet-ids client sent = {:?}", closed.ids_sent());
                let initial = closed.send_ack(&mut server);
            }
        }
    }
}
```

Figure 6: A segment of code that uses the client API.

- The only method in the client API not shown in the state diagram is `ids_sent`. The method is accessible in all states except the Client state, and returns `Vec<u32>`. See below for details.

Use type state to implement the client API in `part2/src/prob2/client.rs` so that every valid API usage should compile and run correctly, and every invalid API usage should not compile. As in Problem 1, each state should correspond to the struct **whose name is the same as the state name**. The signatures of the methods to implement are provided in the starter code; make sure to use them exactly. The reference solution without comments is 66 lines. Detailed instructions:

- To send a signal or a vector of packets to a server, you should call `internal_send_sig(...)` or `internal_send_pkts(...)`, respectively, provided in `part2/src/prob2/server.rs`. Each of the methods `send_*` should call either `internal_send_sig(...)` or `internal_send_pkts(...)` **exactly once**.
- The `ids_sent` method should return `ids`, the ids of all the packets that have been sent by the client and successfully received by a server since the most recent Initial state. The order of ids in `ids` is unimportant.
- You can assume there exist at most one server and at most one client.
- To test your implementation, change the current directory to `part2/`, and run `cargo test prob2::valid -- --show-output` and `cargo test prob2::invalid -- --show-output` as in Problem 1.
  - The last (or second last) line of the test command's output tells you how many test cases you fail.
  - Make sure that the files `part2/wip/client_*.stderr` generated by the invalid test cases are identical to the files `part2/wip/expected_client_*.stderr` which we provided, aside from line & column numbers that depend on your implementation (i.e. in `$DIR/client.rs:[line]:[col]`, line and col can be anything).
  - You may encounter the following warning: "warning: Hard linking files in the incremental compilation cache failed." If so, please ignore them.
- The test cases are in `part2/tests/prob2/mod.rs`. Note again that the test cases are incomplete—it is your task to ensure that valid API usages should compile and invalid ones should not.

Problem 3: `send_all` Function

The goal of this problem is to implement, using the client API from Problem 2, a function `send_all` that takes a server `server` and a vector of packets `pckts` and achieves the following:

- Send all of `pckts` to `server` successfully. The order in which `server` receives the packets is unimportant, but `server` should receive each packet in `pckts` exactly once.
- Close a connection to `server` successfully. That is, a client used to send `pckts` to `server` should be in the Initial state when it is destructed.

Remember that `server` can fail to receive some signals and packets sent by a client. However, you can assume that `server` will receive all signals and packets sent by a client, once it fails to receive a signal or a packet N times in total (where $N = 100$ in the starter code). Hence you can repeatedly send signals and packets to `server` until they are successfully received by `server`, because this is guaranteed to terminate.

Use the client API from Problem 2 to implement the function `send_all` in `part2/src/prob3/sendall.rs`. The reference solution without comments is 33 lines. Detailed instructions:

- You can assume there exist at most one server and at most one client.
- To test your implementation, change the current directory to `part2/`, and run `cargo test prob3::valid -- --show-output`.
  - The last (or second last) line of the test command's output tells you how many test cases you fail.
  - You may encounter the following warning: "warning: Hard linking files in the incremental compilation cache failed." If so, please ignore them.
- The test cases are in `part2/tests/prob3/mod.rs`. Note again that the test cases are incomplete.

Submission

- Regardless of whether you are working in a pair, navigate to Canvas → People, then click on the "hw5-groups" tab and sign up for a hw5 group. If you are working with a partner, make sure you both sign up for the same group.
- Edit `README.txt` to include your student ID number (the 8-digit number) and the student ID number of your partner (if you are working in a pair).
- Generate `solution.tar.gz` by running `python3 submit.py` and upload the tarball file to Canvas.
- **Make sure the script gives no errors or warnings.**
- **Make sure the tarball file consists precisely of {`cart,client,sendall`}`.rs` and `README.txt`.**