Project: Writing an Asynchronous IO Library in Rust (v2.4)

Due date: December 1 (Wednesday) at 11:59pm

1 Overview

The project consists of three parts:

1. Writing building blocks to support asynchronous IO operations (Section 2 – 3).
2. Build a session type library that type checks in the usage of communication protocols between a server and a client (Section 4).
3. Combine 1. and 2. to create an asynchronous communication library (Section 5).

Important notes:

- Because this project relies on the Linux `epoll` API, the code must be run in a Linux environment, such as on Myth. Instructions for remote development via SSH are included in Section 6: Q&A.
- You will use **Rust 1.47.0**, the same version as HW5, which is already installed on the Myth machines. Install rustup 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 (...)`.
- Please read "The Rust Programming Language" book (`https://doc.rust-lang.org/book/`), especially Chapters 1 – 11, 15, 17, 18 to become familiar with Rust.
- We have compiled a tutorial and hints of the problems here.
- You are optionally allowed to work with a partner for this project. One member of each pair should submit the project on Canvas (instructions at the bottom).

2 Futures

There is continuing debate in the programming language community about how to represent asynchrony. Most imperative languages assume a sequential model of computation, where there is only a single thread of control, and only one kind of work is done at a time.

An old and very common model of asynchrony is threads, where computation is launched on independent threads of control, often synchronized through constructs like barriers, mutexes, and condition variables. Threads have been used for decades and because they are so popular there are dedicated hardware resources in modern processors for supporting threads.

However, there are also drawbacks to using threads:

- Threads can be expensive: For most programming languages, context switches between threads is slow as threads need a significant amount of state. Standard thread implementations don't scale beyond thousands of concurrent threads.
- Synchronization between threads is difficult: Synchronization between threads is tricky and notoriously difficult to debug. Even worse, race conditions may cause memory corruption, which could lead to security issues.

In this project, we will explore how to do concurrency **without using threads**.

2.1 Futures in Rust

Futures are objects that represent work to be completed at some point in the future. Let's look at the definition of the `Future` trait in rust:

```rust
pub enum Poll<T> {
    Ready(T),
    Pending,
}

trait Future {
    type Output;
    fn poll(self: Pin<&mut Self>, cx: &mut Context<'_>) -> Poll<Self::Output>;
}
```

Ignore `Pin` and `Context` for now. This definition tells us a `Future` is any object that produces an `Output` through a polling interface, where a future returns `Pending` if the output is not ready, and `Ready(T)` if the future is completed. A call to `poll` should return quickly and never block. So if a future returns `Pending` because it has not completed its task, we can suspend it and switch to other futures.

Async Functions

A function that returns a future is called an **asynchronous function**, or async function for short. Here is an example of an async function in rust:

```rust
struct MyFuture;

impl Future for MyFuture {
    type Output = i32;
    fn poll(/*...*/) -> Poll<Self::Output> {
        /* ... */
    }
}

fn do_something_async() -> MyFuture {
```

```rust
    /* ... */
}
```

Rust also provides language syntax for writing async functions so you do not need to construct a custom struct every time. For example:

```rust
async fn do_something_async() -> i32 {
    if input() > 0 {
        let result = do_another_thing().await;
    }
}
```

We use `async` to mark the function as a asynchronous one, and use `.await` on a future to wait for it to complete before proceeding to the next line. You can imagine the rust compiler automatically translate the code to the previous one, create a state machine `MyFuture` that represent the control flow of what you have written, and run it inside the `poll` function. You will think more about how this can be done in the next section.

See here for more details on the syntax.

2.2 Future combinators

In theory, each async IO library could come up with its own future types. One of the benefits of having a common `Future` trait in the standard library is that anyone can write extensions to the trait, and it can be used by any implementation.

The module `futures::future` contains functions that construct two basic leaf futures, one that completes immediately and one that never completes:

1. `pub fn ready<T>(val: T) -> impl Future<Output = T>`: Returns a future f that is immediately ready. That is, f will produce `Ready(val)` when polled. We have implemented this one for you.
2. `pub fn pending<T>() -> impl Future<Output = T>`: Returns a future that never completes. That is, it will produce `Pending` every time it is polled. **Implement `poll` for `Pending<T>`.**

Existential Type

The syntax `impl Trait` represents an existential type, which means that it could be any type that implements trait `Trait`.

- From the view of clients *using* the returned type `impl Future<Output = T>`, other than it being a type that implements `Future<Output = T>`, no additional information can be assumed for this type.
- From the view of operations *implementing* this type, they are allowed to swap in any type that implements `Trait`.

Thus, if we require you to implement an async function: `fn async_fun() -> impl Future<Output = T>`, you may return an async block or any type that implements `Future`. You can even change the signature to the following variants:

```rust
fn async_fun() -> MyFuture; // For some MyFuture: Future<Output = T>.
async fn async_fun() -> T;
```

The file should also contain these following future combinators in the extension trait `FutureExt` for `Future`. The extension trait is defined as

```rust
pub trait FutureExt: Future {
```

```rust
  /* ... */
}
impl<F: Future> FutureExt for F {}
```

You can assume `self` implements `Future` when developing these combinators.

1. ```rust
   fn map<FN, T>(self, fun: FN) -> impl Future<Output = T>
   where Self: Sized, FN: FnOnce(Self::Output) -> T;
   ```

   Map the result type of future `self` into a new type `T` using the function `fun`. **Implement `poll` for `Map<F, FN, T>`.**

2. ```rust
   fn flatten(self) -> Flatten<Self>
   where Self: Sized, Self::Output: Future + Sized;
   ```

   Flatten the future `self` by waiting for it to return another future and then run that future. This function is already implemented for you.

3. ```rust
   fn then<FN, F>(self, fun: FN) -> impl Future<Output = F>
   where Self: Sized, FN: FnOnce(Self::Output) -> F, F: Future + Sized;
   ```

   Run `self` until completion, then pass the result to function `fun` to get a new future `F`, then run the future `F`. This is used to chain the computation of two futures.

4. ```rust
   fn join<F>(self, f: F) -> impl Future<Output = (Self::Output, F::Output)>
   where Self: Sized, F: Future + Sized;
   ```

   Run `self` and `f` concurrently until both of them complete and return their result in a pair. Notice that you **must** run them concurrently instead of, say, running `self` first until completion and then `f`. **Implement `poll` for `Join<F1, F2>`.**

5. ```rust
   fn select<F>(self, future: F) ->
       impl Future<Output = Either<(Self::Output, F), (Self, F::Output)>>
   where Self: Unpin + Sized, F: Future + Unpin + Sized;
   ```

   Run `self` and `f` concurrently until **one** of them completes, then return the result. If the one that completes first is `self`, return `Either::Left((val, future))` where val is the result from `self`, else return `Either::Right((self, val2))`, where `val2` is the result returned by `future`. In both cases, the returned pair consists of a value and a future. **Implement `poll` for `Select<F0, F1>`.**

When implementing these futures, you may assume that once the future completes, `poll` will never be called again. **You must also follow this rule**.

Notice that currently, Rust does not support existential types in trait functions, so the existential types above are there just to indicate that you can swap in your own types, as long as they implement `Future`.

Pin

The `self` passed into the `poll` function is a `Pin` to the future.

```rust
fn poll(self: Pin<&mut Self>, cx: &mut Context<'_>) -> Poll<Self::Output>;
```

`Pin` is a type of **smart pointer** that guarantees the underlying object cannot be moved. We need this guarantee to ensure memory safety because a future can have self-references. Consider the following example:

```rust
async fn async_fun() {
    let x = 4;
    let y = 2;
    let z = if condition() { &x } else { &y };
    let val = another_async_fun(z).await;
```
```

```
}
```

Now, when this future is waiting for `another_async_fun` to complete, it will need to store `x`, `y`, `z` to the future object itself (futures do not have their own stacks). You can imagine that the Rust compiler constructs the following future struct for you:

```rust
struct __GeneratedFutureObj {
    x: i32,
    y: i32,
    z: &i32,
    // ...
}
```

In this case, `z` will point to the address of either `x` or `y`. However, if the object is moved to a new location, `z` will point to an invalid location! Thus, we need to use `Pin` to ensure that the future object is not moved.

We will skip the details for `Pin` object and focus on what you need to implement this project. We used an external library `pin-project`. If you have a pin `pin: Pin<&mut T>`, where `T` is defined as:

```rust
#[pin_project]
struct T {
    x: T1,
    #[pin] y: T2,
}
```

You can then access the fields by calling `pin.project()`. `pin.project()` will project the `Pin` into subfields and return a type like:

```rust
struct TProj {
    x: &mut T1,
    y: Pin<&mut T2>,
}
```

You do not need to define the structs that require pinning yourself. We defined all those structs for you. You just need to know how to use it. See the rust guide for more details.

[0] In Rust's term, these types implement `Unpin`

3 Async IO Runtime

3.1 Executor

While futures define independent units of work, we still need an underlying runtime that can execute them concurrently. This is the idea of an executor, or future-executing engine.

Here is the definition of the executor:

```
impl Executor {
    pub fn new() -> Self;
    pub fn handle(&self) -> Handle;
    pub fn spawn<F>(&self, future: F) where F: Future<Output = ()> + 'static;
    pub fn run(&mut self);
}

impl Handle {
    pub fn spawn<F>(&self, future: F) where F: Future<Output = ()> + 'static;
}
```

Users will first create their futures, use `spawn` to add them into the executor, and call `run`. `run` will run these futures concurrently until all the futures are completed. `handle` will return a `Handle` that can be passed into a future to spawn more futures asynchronously.
Please read through the whole of section 3 before staring to code.

3.2 Async IO Futures

Our library will also provide these three types of futures.

1. Networking IO (`futures::io`): `AsyncListener` and `AsyncStream`. These two structs will resemble their synced counterpart `TcpListener` and `TcpStream` in Rust's standard library. They need to have the following methods:

   ```
   impl AsyncStream {
       pub fn connect<A: ToSocketAddrs>(addr: A) -> io::Result<Self>;
       pub fn read(&mut self, buf: &mut [u8]) -> impl Future<Output = io::Result<usize>>;
       pub fn read_exact(&mut self, buf: &mut [u8]) -> impl Future<Output = io::Result<()>>;
       pub fn write(&mut self, buf: &[u8]) -> impl Future<Output = io::Result<usize>>;
       pub fn write_all(&mut self, buf: &[u8]) -> impl Future<Output = io::Result<()>>;
   }

   impl AsyncListener {
       pub fn bind<A: ToSocketAddrs>(addr: A) -> io::Result<Self>;
       pub fn accept(&self) -> impl Future<Output = io::Result<(AsyncStream, SocketAddr)>> + '_
   }
   ```

   These functions should match the logic of `TcpStream::connect`, `TcpStream::read`, `TcpStream::read_exact`, `TcpStream::write`, `TcpStream::write_all`, `TcpListener::bind`, and `TcpListener::accept`, with the exception that they are asynchronous and should never block. See the code, [TcpSream](), and [TcpListener](), for more details. You don't need to implement most of the underlying logic of these functions. Rather, you only need to wrap the synced version. For instance, for `AsyncStream::read`, you will call `TcpStream::read`, and return `Poll::Ready` or `Poll::Pending` depending on the outcome.

Error Handling

We will not be strict in error handling. You can assume all **normal** system operations (e.g., system calls) will not fail, except for errors from IO operations (because that could indicate, say, the server terminates the connection). These error will correspond to `std::io::Result` class in Rust.

For those operations that will not or are assumed not to fail, you can simply call `unwrap` or `expect` on the `Result` value. For `std::io::Result`, please make good use of the question mark operator `t`. Specifically, `x = expr?` is roughly equvalent to

```
// If expr is an Result
x = match expr {
    Ok(val) => val,
    Err(err) => return err,
}
// If expr is an Option
x = match expr {
    Some(val) => val,
    None => return None,
}
```

2. A timer class (`futures::timer`).

```
pub struct Timer;
impl Timer {
    pub fn new(dur: Duration) -> Self {
        unimplemented!()
    }
}
```

   The future should complete only after a duration of `dur`, and we only ask you to support a granularity of 100 milliseconds. This means that you only need to check the timers periodically every, say, 50 milliseconds. Technically, if x is the current time when `Timer::new` is called, we expect the future to return in the range of $[x - 100\text{ms}, x + 100\text{ms}]$ when the executor is not busy.

3. Synchronization primitives (`futures::sync`). There is no need to worry about race conditions because we are running only on a single thread where every operation is atomic (so for acquiring a lock, we can simply set `locked = true` or return `Poll::Pending` if it is already locked). Still, it is handy to have synchornization constructs to prevent busy waiting, so that when a future is waiting to acquire a lock the executor can temporarily suspend it until another future releases the resource.

```
impl Condvar {
    pub fn new() -> Self;
    pub fn wait(&self) -> impl Future<Output = ()>;
    pub fn wait_while<F>(&self, mut condition: F) -> impl Future<Output = ()>
    where F: FnMut() -> bool;
    pub fn notify_one(&self);
    pub fn notify_all(&self);
}

impl Mutex {
    pub fn new() -> Self;
    pub fn lock(&self) -> impl Future<Output = MutexGuard<'_>>;
}
```

   - For `Condvar`, `wait` should wait until `notify_one` or `notify_all` is called. These functions should resemble those of std::sync::Condvar.

- For Mutex, lock should wait until no one else is holding the mutex.
  It should resemble std::sync::Mutex. There is no unlock function for Mutex. Instead, implement unlocking in the drop function, which is the destructor for MutexGuard.

3.3 Reactor

When implementing the executor described above, the first thought that comes to mind might be to store futures somewhere inside the executor, and when run is called, loop through each future in a round-robin manner and call poll. Repeat this process until all the futures are resolved. This approach will work; however, the problem is that it suffers from **busy waiting**. That is, if all the futures are blocked, perhaps because they are waiting for IO, we will still poll each of them, get Pending each time, and waste CPU resources.

We solve this problem by using a **reactor**. When a future returns Poll::Pending, it register itself, together with an **event** for which it is waiting, with the reactor. There can be multiple types of events. For this project, we are interested in events for files (which correspond to those TCP streams), timers, and releasing locks.

For file IO events, we use Linux's epoll API. There are two important functions, epoll_ctl and epoll_wait. We use epoll_ctl to register file descriptors that we are interested in. If IO on these file descriptors will block, we call epoll_wait and suspend the main thread. Whenever one of the file descriptors becomes ready, the kernel will wake our process and notify us.

As this part is not the main interest of this project, we have implemented these operations in struct Reactor for you. However, you still need to extend Reactor so it supports timer and lock events.

3.4 Waker

A waker is an object used by the reactor to notify the executor that a future is ready to proceed. Rust has a common interface for Waker, and we need to implement the following functions:

```
unsafe fn clone(data: *const ()) -> RawWaker;
unsafe fn wake(data: *const ());
unsafe fn wake_by_ref(data: *const ());
unsafe fn drop_arc_raw(data: *const ());
```

You could implement these function directly for your waker, but playing with unsafe in Rust might be tricky. We suggest you implement our ArcWake interface instead, and we will wrap those functions for you. The ArcWake is defined as:

```
pub trait ArcWake {
    fn wake(self: &Arc<Self>);
}
```

This struct will be passed to futures inside the Context in the poll function. When wake is called, it should resume a future that was suspended because it was waiting for an IO operation. We will use Arc::clone to make multiple clones of it, so we can store it elsewhere and wake up the future afterward.

Your waker needs to be able to communicate with your executor. When you call Future::poll inside your executor, you should store some information in the waker so that it can notify the executor when wake is called. There are several ways to achieve this.

1. Store a flag active in each future. The executor only poll futures such that active = true. The future can be suspended by setting active = false and the waker wakes it up by setting blocked = false.

2. The executor has a task queue that stores active futures. When wake is called, the waker pushes the future or the task object back to the queue. In this case, the waker could be an object containing the future and a reference to the task queue.

There are plenty of designs with different advantages here. We suggest you to implement the second one. A scaffold is provided to you in `futures::executor`.

Reference Counting

When multiple ownership is needed, you can use an `Rc` or `Arc`. These are reference-counting pointers that will automatically drop the object when the last reference goes out of scope. You call `Rc::clone(x)` or `Arc::clone(x)` to get a shallow copy of `x`.

The only difference between `Rc` and `Arc` is that `Arc` is thread-safe, while `Rc` is not. In this project, you only need to use `Rc` because we are running the whole program in a single thread. The only exception is when you are implementing `ArcWake`, you need to use `Arc` because Rust imposes a rule that `Waker` must be thread safe.

See here for more details.

After implementing a waker, the reactor will start to work. We have already implemented these functions for you:

```
pub(super) fn register_fd_events<T: AsRawFd>(
    object: &T, interest: Interest, waker: Waker) -> Token;
pub(super) fn deregister_waker(id: usize);
pub(super) fn wait_fd_events(dur: Duration);
```

Visibility

Visibility in Rust restricts access to items (functions, structs, etc.):

- `pub` is visible to anyone.
- `pub(super)` is only visible to code in the same (sub)module.
- By default, an item is only visible to the file where it belongs.

You can make any changes to items that are not marked `pub`. This includes `pub(super)` items and fields in a `pub` struct that is not marked `pub` (these are still count as private items). The signature of `pub` items need to be exactly the same, with exceptions that are described in the "existential types" section. Also, while the mutability of a reference is part of the signature, the **binding** is not. So you can change `fn f(self)` to `fn f(mut self)` (but not `fn f(&self)` to `fn f(&mut self)`).

You can use `register_fd_events` to register an event for a future. When there is an event from the file descriptor `fd` (e.g., the stream becomes readable), its waker will be called by the reactor. The function will return a token. When the token is dropped, it automatically deregisters the event, so you need to store the token inside your future. In your executor, call `wait` to suspend until the next event, or a timeout occurs.

Keep in mind that you need to extend this reactor to support timer events mentioned above. How to do it is up to you.

Here is one strategy:

1. Extend the reactor so that it supports registering timer events. For each timer event, store a timestamp and a waker.
2. Each time you return from `wait_fd_events`, check if any timer expires. If one does, call the waker to place the future back in the executor.

Figure 1 is a diagram showing how the executor, the reactor, and futures work together. There are also some runtime semantics that you need to follow:

1. Your future should never block.

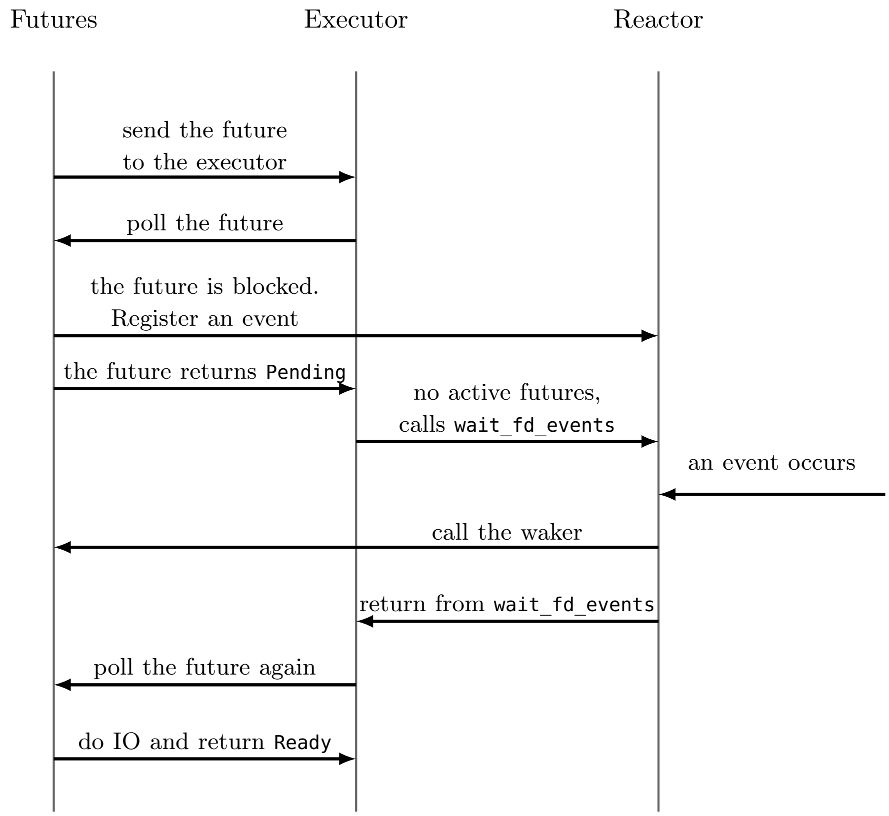

Figure 1: An example showing the interactions between these three pieces.

2. Your executor should never poll a future again once it is complete.
3. When all the futures complete, your executor must return from `Executor::run`.
4. If a future returns `Pending`, your executor should not poll it again unless its waker is called.

When implementing this section, we suggest you proceed as follows:

1. First, implement a busy-waiting executor that simply loops through each future and polls it.
2. Implement those futures mentioned in 3.2.
3. Implement your waker struct.
4. Modify your executor to support wakers.
5. Extend the reactor to support timer events.

Implement the executor module (`futures::executor`).
Implement the three types of future mentioned in 3.2 (`futures::io`, `futures::timer`, `futures::sync`).
Implement your waker type (at the bottom of `futures/executor.rs`).
Extend the reactor module (`futures::reactor`).

4 Session Types

4.1 Definition

In the lecture and HW5, we saw how we could use typestate to describe protocols so that incorrect uses of certain APIs result in compile-time errors. But what about communication protocols that have both client and server sides? In this case, there are additional relationships between the state machines from both sides. For example, in HW5, when a client is sending packets to the server, the server must also be in a state that can receive packets. We will see how we can formalize these additional relationships by using **session types**.

Let's consider the example of an ATM. We can describe a simple protocol for deposits and withdrawals:

1. The client sends their ID to the ATM.
2. The ATM then answers either ok or err.
   - In the first case, the client then proceeds to request either a deposit or withdrawal.
     - For a deposit, the client first sends an amount, then the ATM responds with the updated balance, and the session terminates.
     - For a withdrawal, the client sends the amount to withdraw, then the ATM responds with either ok or err to indicate whether or not the transaction was successful, and the session terminates.
   - If the ATM answers err, the session terminates.

We formalize the ATM's protocol using session types as below:

$$\begin{aligned}
\mathsf{ATM} &= \mathsf{recv}\ id;\ \mathsf{choose}\,\{\ \mathsf{ok} : (\mathsf{ATM}_{\mathsf{auth}}) \mid \mathsf{err} : (\epsilon)\ \} \\
\mathsf{ATM}_{\mathsf{auth}} &= \mathsf{offer}\,\{\ \mathsf{deposit} : (\mathsf{recv\ int};\ \mathsf{send\ int};\ \varepsilon) \mid \mathsf{withdraw} : (\mathsf{recv\ int};\ \mathsf{choose}\,\{\ \mathsf{ok} : (\varepsilon) \mid \mathsf{err} : (\varepsilon)\ \})\ \}
\end{aligned}$$

Session types have four core operations: send/receive, which indicate sending and receiving messages of a particular type from the other party, and choose/offer, which indicate branch points where the host party can choose to enter one of several sub-protocols or offer to let the other party select a sub-protocol.

The key is that there should also be a corresponding session type describing the protocol from the view of the customer (called the "**dual**"), so that both can be ultimately implemented and verified against the appropriate session type. For the previous example, the dual view from the client is:

$$\begin{aligned}
\mathsf{Client} &= \mathsf{send}\ id;\ \mathsf{offer}\,\{\ \mathsf{ok} : (\mathsf{Client}_{\mathsf{auth}}) \mid \mathsf{err} : (\epsilon)\ \} \\
\mathsf{Client}_{\mathsf{auth}} &= \mathsf{choose}\,\{\ \mathsf{deposit} : (\mathsf{send\ int};\ \mathsf{recv\ int};\ \varepsilon) \mid \mathsf{withdraw} : (\mathsf{send\ int};\ \mathsf{offer}\,\{\ \mathsf{ok} : (\varepsilon) \mid \mathsf{err} : (\varepsilon)\ \})\ \}
\end{aligned}$$

One important feature that is missing from this example is recursion. For example, we might want a protocol that the client can keep sending numbers to the server until it chooses to stop. For this, we can represent this behavior with recursive session types, for example:

$$\mathsf{Client} = \mu\,\alpha.\ \mathsf{choose}\,\{\ \mathsf{continue} : (\mathsf{send\ int};\ \alpha) \mid \mathsf{end} : (\varepsilon)\ \}$$

A recursive type $\mu\,\alpha.\ \sigma$ satisfies $\alpha = \sigma$, where α can appear free in σ. Another way of saying the same thing is that $\mu\,\alpha.\ \sigma$ is equal to $\sigma[\alpha := \mu\,\alpha.\ \sigma]$, that is, replacing each occurence of α with $\mu\,\alpha.\ \sigma$.

To make it clear, we formally define the grammar of session type as follows:

$$\begin{aligned}
\sigma \coloneqq\ & \mathsf{recv}\ \tau;\ \sigma \\
\mid\ & \mathsf{send}\ \tau;\ \sigma \\
\mid\ & \mathsf{choose}\ \{\ L : (\sigma_L) \mid R : (\sigma_R)\ \} \\
\mid\ & \mathsf{offer}\ \{\ L : (\sigma_L) \mid R : (\sigma_R)\ \}
\end{aligned}$$

$$
\begin{aligned}
&\mid \varepsilon \\
&\mid \mu\,\alpha.\ \sigma \\
&\mid \alpha
\end{aligned}
$$

And the dual type $\overline{\sigma}$ of σ is defined as follows:

$$
\begin{aligned}
\overline{\mathsf{send}\ \tau;\ \sigma} &= \mathsf{recv}\ \tau;\ \overline{\sigma} \\
\overline{\mathsf{recv}\ \tau;\ \sigma} &= \mathsf{send}\ \tau;\ \overline{\sigma} \\
\overline{\varepsilon} &= \varepsilon \\
\overline{\mathsf{choose}\ \{\ L : (\sigma_L) \mid R : (\sigma_R)\ \}} &= \mathsf{offer}\ \{\ L : (\overline{\sigma_L}) \mid R : (\overline{\sigma_R})\ \} \\
\overline{\mathsf{offer}\ \{\ L : (\sigma_L) \mid R : (\sigma_R)\ \}} &= \mathsf{choose}\ \{\ L : (\overline{\sigma_L}) \mid R : (\overline{\sigma_R})\ \} \\
\overline{\mu\,\alpha.\ \sigma} &= \mu\,\alpha.\ \overline{\sigma} \\
\overline{\alpha} &= \alpha
\end{aligned}
$$

4.2 Session Types in Rust

We implement session types in Rust. We define these types as

```
struct Send<T, S>(/* ... */);
struct Recv<T, S>(/* ... */);
struct Offer<Left, Right>(/* ... */);
struct Choose<Left, Right>(/* ... */);
struct Close; // equivalent to ε
struct Rec<S>(/* ... */); // equivalent to μα. #1.
struct Var<N>(/* ... */); // equivalent to α.
```

It should be clear how these definitions map to the session type, except for the recursive ones.

In the definition of `Rec`, instead of using identifiers to represents variables, we will use De Bruijn indices. For example, the following recursive session type

$$\mu\,\alpha.\ \mathsf{send\ int};\ \mu\,\beta.\ \mathsf{choose}\ \{\ \mathsf{left} : (\alpha) \mid \mathsf{right} : (\beta)\ \}$$

is represented by

$$\mu.\ \mathsf{send\ int};\ \mu.\ \mathsf{choose}\ \{\ \mathsf{left} : (1) \mid \mathsf{right} : (0)\ \}$$

Each number n represents the variable defined at the n-th outer-scope from the current scope. In other words, there will be n other μs that are between the variable and the μ that defines this variable.

We will need a way to represent numbers which we can use at compile time[1]. So as a work around, we will use the following type:

```
struct Zero;    // zero
struct Succ<N>; // succ(N)
```

You should be very familiar with this representation of numbers by now! `Zero` will represent a zero and `Succ<N>` will represent $N + 1$.

Your task is to implement the `SessionType` trait for these structs. The `SessionType` has the following definitions:

```
trait SessionType {
    type Dual: SessionType;
}
```

[1] In other languages like C++, we could use const generic. Unfortunately, const generic in Rust has not yet stabilized.

The trait mostly acts as a marker, indicating that the type is a session type. The only requirement we impose on this trait is that it has a dual.

Implement session types (`session::types`).

5 Putting it All Together

Now, we will combine the previous parts to create an asynchronous communication library that uses session-type to ensure the client and the server use the protocol correctly.

5.1 Session Type Channels

Here is how a user will use your session type channel API:

```
type Client = Rec<Choose<Send<u64, Var<Zero>>, Recv<String, Close>>>;
type Server = Client::Dual;

async fn client() -> io::Result<()> {
    let channel = Channel::<Client>::connect(addr);
    let channel = channel.step();
    let channel = channel.choose_left().await?;
    let channel = channel.send(42).await?;
    let channel = channel.step();
    let channel = channel.choose_right().await?;
    let (channel, val) = channel.recv();
    // ...
}

async fn server() -> io::Result<()> {
    let listener = Listener::<Server>::bind(addr);
    let (channel, _) = listener.accept().await?;
    let channel = channel.step();
    loop {
        match channel.offer().await? {
            Left(channel) => {
                let (channel, value) = channel.recv().await?;
                println!("Received {} from client.", value);
                // ...
            }
            Right(channel) => {
                let channel = channel.send("hello!".to_owned()).await?;
                // ...
            }
        }
        // ...
    }
}
```

Your goal is to implement the `Channel` and the `Listener` struct. Specifically, `Channel` will be a type supporting session type states with the following definition:

```
struct Channel<S: SessionType, Env> {
  /* ... */
}
```

`S` refers to the current type state of the session, and `Env` will be a tuple of session types that we use to deal with recursive session types.

Different session types should implement different methods, specifically:

- For `Send<T, _>`, there should be a function

  ```
  pub fn send(self, val: T) -> impl Future<Output = io::Result<Channel</* ... */>>>
  ```

The function will send a `val` to the channel. Recall that this signature is equivalent to

```
pub async fn send(self, val: T) -> io::Result<Channel</* ... */>>
```

`send` takes a `val` and sends it. It should call `BytesRepr::serialize` and send the whole byte array to the stream. See the next section for more details.

- For `Recv`, there should be a function

  ```
  pub fn recv(self) -> Future<Output = io::Result<(Channel</* ... */>, T)>>
  ```

  that will receive a value from the channel. It should call `BytesRepr::deserialize` to get the value from the bytes. See the next section for more details.

- For `Offer`, there should be a function

  ```
  pub fn offer(self) ->
      Future<Output = io::Result<Either<Channel</* ... */>, Channel</* ... */>>>>
  ```

  that waits for the other side to choose. Notice the `Either` type here. If the other side chooses left, return `Either::Left(channel)`, else return `Either::Right(channel)`.

- For `Choose`, there should be two functions

  ```
  pub fn choose_left(self) -> Future<Output = io::Result<(Channel</* ... */>)>>
  pub fn choose_right(self) -> Future<Output = io::Result<(Channel</* ... */>)>>
  ```

  that each chooses one option.

- For `Close`, there will be no functions.

For the recursive types, things are trickier. Both `Rec` and `Var` will only have a function called `step`:

```
pub fn step(self) -> Channel</* ... */>
```

Unlike those functions for non-recursive session types, there is no real work need to be done (sending/receiving messages) in the `step` function, so the return type will simply be a `Channel</* ...  */>` instead of a future. Now, let's look at what the return type should be. For `Rec`, we want to store the current type in the environment `Env`. Recall that the environment is a list of session types, which will be represented using pairs in Rust. For a list $S_0, S_1, \ldots, S_n$, we represent it as `(S0, (S1, (..., (Sn, ()))))`. So the definition of `step` in `Rec` should be:

```
impl<S: SessionType, Env> Channel<Rec<S>, Env> {
    fn step(self) -> Channel<S, (S, Env)>;
}
```

We leave the definition of `Var<N>` to you. It will be more involved than `Rec` because you will need to pop `Env` N times to get the type.

Specifically, if the environment is the list $S_0, S_1, \ldots, S_n$ and we encounter `Var<M>`, we need to get S_M from the list (which will be the next session type to proceed) and restore the environment to be $S_M, \ldots, S_n$.

To achieve this, you will implement the following recursion:

$$\mathsf{Pop}(L, N).\mathsf{List} = \begin{cases} L, & \text{if } N = 0, \\ \mathsf{Pop}(T, N-1).\mathsf{List}, & \text{otherwise, let } (H, T) = L \end{cases}$$

$$\mathsf{Pop}((H, T), N).\mathsf{Head} = \begin{cases} H, & \text{if } N = 0, \\ \mathsf{Pop}(T, N-1).\mathsf{Head}, & \text{otherwise,} \end{cases}$$

using Rust's type system! Specifically, we will define a trait:

```
pub trait Pop<N: Number> {
    type List;
```

```
    type Head;
}
```

so that `<L as Pop<N>>::List` and `<L as Pop<N>>::Head` are the results of $\mathsf{Pop}(S, N).\mathsf{List}$ and $\mathsf{Pop}(S, N).\mathsf{Head}$, respectively.

Finally, for the `Listener` struct, it should be very easy. There is only one function you need to implement:

```
pub fn accept(&self) -> impl Future<Output = io::Result<(Channel<S, ()>, SocketAddr)>>;
```

Implement the recursion using types (`session::types`).

5.2 Sending the Data

The final issue is how to actually send data through TCP streams. In particular, we will provide you a trait:

```
pub trait BytesRepr: Sized {
    const LEN: Option<usize>;
    fn serialize(self) -> Vec<u8>;
    fn deserialize(bytes: &[u8]) -> Option<Self> {
}
```

`serialize` will return a byte array representation of the object. `deserialize` will try to form an object from the byte array, and return `None` if there is an error. The implementation guarantees that `deserialize(&serialize(x)) == Some(x)` always holds.

- For types that have a fix length (e.g., `u32`, `u64`), `len` will be `Some(len)`. When sending this kind of type we only need to send its representation, and when receiving it, we know exactly how many bytes we need to read.
- For types that have a variable length (e.g., `Vec`, `String`), `len` will be `None`. When sending this kind of type, first, we send its length as an `u32` (which is 4 bytes) to the stream and then send its representation. When reading it, we will do the reverse. This is more complicated because we do not know the length in advance. Specifically, you should:
    1. Read an `u32` from the stream to get the length `len`.
    2. Create a buffer with length equal to `len`. Set the first four bytes to be `len` by using `set_len`. (We need to set the first four bytes because `BytesRepr::deserialize` will expect this value.)
    3. Read the rest of the data from the stream.

The inner type of `Recv` and `Send` should have this trait bound, so you can use these function when implementing `recv` and `send`.

Implement `Channel` and `Listener` structs (`session::channel`).

6 Q&A

1. **What version of rust should I use?**
   `1.47.0`.

2. **Can I include other libraries?**
   No, you are not allowed to use any external libraries other than those already included in `Cargo.toml` (namely, `nix`, `pin-project`, and `either`). Or else you can simply wrap other future libraries!

3. **Where do I need to make changes?** Every `unimplemented!()`, every type that is `Never`, and a few places that have `TODO`s. You might need to change other places depending on your design of section 3.

4. **How do I test my code?**
   You can run `cargo test`. We provide some test cases, and there will also be some hidden test cases. Notice that `cargo test` stops at the first error. To run all tests, run `cargo test --no-fail-fast`. See here on details about how to run and write tests.

5. **How do I generate documentations?**
   Run `cargo doc --document-private-items`. Also, `cargo doc --document-private-items --open` will (sometimes) open the browser for you.

6. **What is the recommended development environment?**
   You can only run the code on Linux because we are using `epoll` API (we can port it to other OS using `IOCP` or `kqueue`, but that is left to you as an exercise). For the editor, choose those that support language clients and rust-analyzer. A good choice might be Visual Studio Code with remote development extension (see here) so you can develop the code locally while running on Myth machines.

7. **Do we need to tidy our code and write documentation?**
   No, that is not needed. But doing so might assist you in debugging your code. We suggest you run Rust's linter called clippy.

7 Submission

- Regardless of whether you are working in a pair, navigate to Canvas → People, then click on the "proj-rust-groups" tab and sign up for a proj-rust group. If you are working with a partner, make sure you both sign up for the same group.
- Edit `Cargo.toml` to include your SUNet ID (the username of your Stanford email) and the SUNet ID 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.**