Lecture 11

Async/Await

Concurrency Models

OS Threads

This is the concurrency model we have been using so far. We spawn in a `std::thread`, which under the hood is an OS-level operation.

For small tasks (such as downloading a single file), spawning an entire *thread* seems overkill.

Pros:

- Simple to use

Cons:

- Each time we spawn a thread, there is a performance cost (syscall, context switch)

```rust
fn get_two_sites() {
    // Spawn two threads to do work.
    let thread_one = thread::spawn(|| download("https://www.foo.com"));
    let thread_two = thread::spawn(|| download("https://www.bar.com"));

    // Wait for both threads to complete.
    thread_one.join().expect("thread one panicked");
    thread_two.join().expect("thread two panicked");
}
```

Coroutines (a.k.a. “Green Threads”)

This is the concurrency model used by languages like Java, Python, Go, and Lua.

Instead of using OS-level threads, the language runtime supports the creation of cheap “fake” threads. The runtime then decides which “thread” to execute at any given time.

Pros:

- Cheaper to spawn than OS threads
- Simple and easy to use

Cons:

- Requires a language runtime, which is unsuitable for a systems language
    - Rust prefers “zero-cost abstractions”

```java
public class SingleThreadExample {
    public static void main(String[] args) {
        NewThread t = new NewThread();
        t.start();
    }
}
```

Event-driven programming (callbacks)

This concurrency model is frequently seen in JavaScript (although they also support async/await syntax).

It works by passing "callback functions" as arguments.

```
function func() {
    console.log("line 1");
    setTimeout(() => {
        console.log("line 2");
    }, 2000);
    console.log("line 3");
}
func();
```

Pros:

- Very performant

Cons:

- Verbose, nonlinear control flow
- Hard to debug

Async/await syntax

We introduce two new keywords:

- `async`
- `.await`

We use `async` to mark a function as "asynchronous", and we use `.await` to *await* the execution of another `async` function (and do other work in the meantime).

Pros:

- Writing asynchronous code "feels" like writing synchronous code

Cons:

- Also requires a runtime (more later…)
- Leaky abstraction

```rust
#[tokio::main]
async fn main() -> Result<()> {
    let mut client = client::connect("127.0.0.1:6379").await?;
    client.set("hello", "world".into()).await?;
    let result = client.get("hello").await?;
    println!("got value from the server; result={:?}", result);
    Ok(())
}
```

Async/await

What does it mean for a function to be async?

Consider the following function:

```rust
async fn fetch(url: &str) -> Option<Response>;
```

What does it mean for a function to be `async`?

Consider the following function:

```
async fn fetch(url: &str) -> Option<Response>;
```

Suppose we call `fetch`:

```
let response = fetch("https://www.foo.com");
```

What does it mean for a function to be `async`?

Consider the following function:

```
async fn fetch(url: &str) -> Option<Response>;
```

Suppose we call `fetch`:

```
let response = fetch("https://www.foo.com");
```

What is the type of `response`?

```
let response: impl Future<Output = Option<Response>>
```

What does it mean for a function to be `async`?

Consider the following function:

```
async fn fetch(url: &str) -> Option<Response>;
```

Suppose we call `fetch`:

```
let response = fetch("https://www.foo.com");
```

What is the type of `response`?

```
let response: impl Future<Output = Option<Response>>
```

What is a `Future`?

- It's like a "promise" of some *future* value that does not yet exist.

What does it mean for a function to be `async`?

Consider the following function:

```
async fn fetch(url: &str) -> Option<Response>;
```

Suppose we call `fetch`:

```
let response = fetch("https://www.foo.com");
```

What is the type of `response`?

```
let response: impl Future<Output = Option<Response>>
```

What is a `Future`?

- It's like a "promise" of some *future* value that does not yet exist.

How do we get the value of a `Future`?

- We need to `.await` it.

Async/await syntax is a leaky abstraction

```
async fn app() {
    let response = fetch("https://www.foo.com").await;
}
```

Async/await syntax is a leaky abstraction

```
async fn app() {
    let response = fetch("https://www.foo.com").await;
}
```

In order to use `.await`…

Async/await syntax is a leaky abstraction

…we need to mark our function async

```
async fn app() {
    let response = fetch("https://www.foo.com").await;
}
```

In order to use .await…

Implementation & Rationale

Why?

When we call `.await`, we don't want to **block** the current thread. Therefore, we also need to change the *calling* function to return a `Future`.

In reality, `async` functions are less like *functions* and more like **state machines** built up by *composing together* other `async` "state machines" (i.e. functions).

Notice that `Future` is a `trait`. That means that each `Future` has its own **state**.

Each time we `poll` a future, it advances its **state** as much as possible (until the future is `Ready`).

```
pub trait Future {
    type Output;

    fn poll(...) -> Poll<Self::Output>;
}
```

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

So how do we actually call `async` code?

So the question remains, how do we *call* `async` functions? *Eventually*, we will need to call them "synchronously" from our `main` function.

- Async functions actually return a `Future<T>`, when really we just care about the `T`
- We can't get the `T` by `.await`ing it, because then we would need to make our function `async`

This is where **executors** come in.

At a high level, an *executor* intelligently calls `poll` on our `Future`s until they are `Poll::Ready`. This is the "runtime" component of async/await.

- Rust doesn't actually provide any executor!
    - Popular crates like `tokio` provide this (in fact, `tokio` is a de-facto standard)
- In this way, `async/await` syntax is a "zero-cost" abstraction (or at least low-cost)

How does the executor know when to `poll` futures?

This is managed by something called a `Waker`, which provides a `wake` function that tells the executor the future is ready to make progress (the details of this are not important).

Importantly, futures **will not make progress** unless you `.await` them ("lazy" futures). This is in contrast to languages like JavaScript (which has "eager" futures).

```rust
async fn say_hello() {
    println!("Hello from say_hello");
}

#[tokio::main]
async fn main() {
    say_hello();
    println!("Hello from main");
}
```

```
➜  rust-test cargo run
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.01s
     Running `target/debug/rust-test`
Hello from main
➜  rust-test
```

Running `Futures` together

The nice thing about futures is that we can compose them together to make new futures.

The `join!` macro lets us await on futures by running them concurrently. It also implicitly calls `.await` for you.

This is in contrast to awaiting the futures *in sequence*, which would take 5 seconds instead of 3 in this particular (contrived) instance.

```rust
use std::time::Duration;
use tokio::join;

async fn say_hello_1() {
    tokio::time::sleep(Duration::from_secs(2)).await;
    println!("Hello from function 1");
}

async fn say_hello_2() {
    tokio::time::sleep(Duration::from_secs(3)).await;
    println!("Hello from function 2");
}

#[tokio::main]
async fn main() {
    join!(say_hello_1(), say_hello_2());
}
```

```
→  rust-test cargo run
   Compiling rust-test v0.1.0 (/home/alexander/rust-test)
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.55s
     Running `target/debug/rust-test`
Hello from function 1
Hello from function 2
→  rust-test
```

Other `tokio` features

Tokio has other ways of dealing with tasks and futures

- `spawn` lets you spawn “green threads”, although joining them requires the use of `.await`.
- `try_join!` lets you early return if an error is encountered from one of the `Futures`.
- `select!` returns the *first* branch that completes, rather than waiting for all of them.

Tokio also provides additional channels on top of `std::sync::mpsc`

- `tokio::sync::mpsc` – same as standard library (multiple producer, single consumer)
- `tokio::sync::broadcast` – multiple senders & receivers
- `tokio::sync::oneshot` – used to send a single value from one sender to one receiver
- `tokio::sync::watch` – a single sender send values to several receivers, only latest value kept

Considerations & Issues

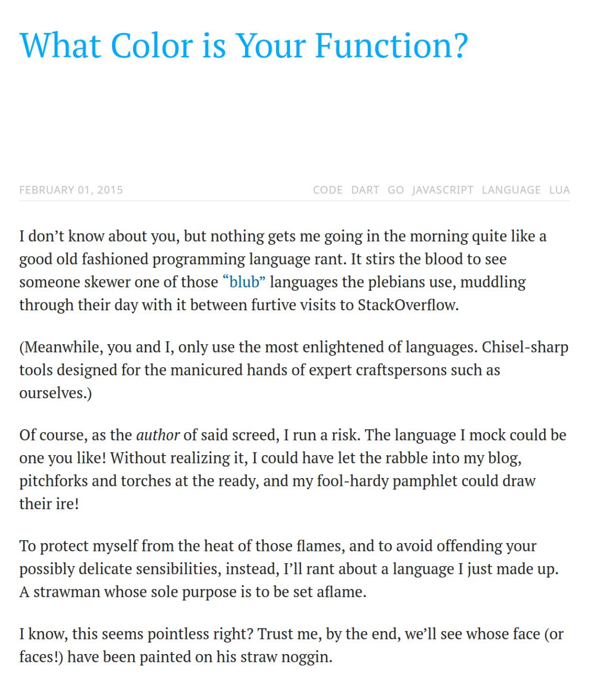
Function coloring

`Async/await` is often considered a leaky abstraction. We can call `sync` code from `async` code, but we *cannot* call `async` code from `sync` code. Furthermore, it is bad practice to call *blocking* `sync` code from async `async`, because then our executor gets hung up, so we can't do other work in the background.

There are three solutions to this dilemma:

1. Only use sync code (but then we don't get `async` features)
2. Use only async code (but can't call from a `sync` context)
3. Make two different versions of every function

This issue is illustrated to great effect in the popular article [“What Color is Your Function?”]() by Bob Nystrom.



What Color is Your Function?

FEBRUARY 01, 2015 CODE DART GO JAVASCRIPT LANGUAGE LUA

I don't know about you, but nothing gets me going in the morning quite like a good old fashioned programming language rant. It stirs the blood to see someone skewer one of those “blub” languages the plebians use, muddling through their day with it between furtive visits to StackOverflow.

(Meanwhile, you and I, only use the most enlightened of languages. Chisel-sharp tools designed for the manicured hands of expert craftspersons such as ourselves.)

Of course, as the *author* of said screed, I run a risk. The language I mock could be one you like! Without realizing it, I could have let the rabble into my blog, pitchforks and torches at the ready, and my fool-hardy pamphlet could draw their ire!

To protect myself from the heat of those flames, and to avoid offending your possibly delicate sensibilities, instead, I'll rant about a language I just made up. A strawman whose sole purpose is to be set aflame.

I know, this seems pointless right? Trust me, by the end, we'll see whose face (or faces!) have been painted on his straw noggin.

Two versions of the same code

Inevitably, we have landed on the third option, so we end up with entire crates like `async_std`, which are *virtually* identical to the standard library but every function is “colored” with the `async` decorator.

```
use async_std::io::Stdin;
use std::io::Stdin;

use async_std::net::TcpStream;
use std::net::TcpStream;
```

Crate async_std

source · [-]

[-] Async version of the Rust standard library

`async-std` is a foundation of portable Rust software, a set of minimal and battle-tested shared abstractions for the broader Rust ecosystem. It offers std types, like `Future` and `Stream`, library-defined operations on language primitives, standard macros, I/O and multithreading, among many other things.

`async-std` is available from crates.io. Once included, `async-std` can be accessed in `use` statements through the path `async_std`, as in `use async_std::future`.

Keyword generics

There have been efforts made to resolve this issue, such as the Keyword Generics Initiative, which proposes the addition of `?async` syntax to mark a function as “maybe async”.

The function can be used from both `async` and non-`async` contexts, where `.await` becomes a no-op. A similar proposal is being made for a `?const` keyword.

This could possibly solve the issue but there is also fear of increasing the complexity of the language. We do not want to end up like C++, which has 97 different keywords and a conglomerate of features.

```rust
trait ?async Read {
    ?async fn read(&mut self, buf: &mut [u8]) -> Result<usize>;
    ?async fn read_to_string(&mut self, buf: &mut String) -> Result<usize> { ... }
}

/// Read from a reader into a string.
?async fn read_to_string(reader: &mut impl ?async Read) -> std::io::Result<String> {
    let mut string = String::new();
    reader.read_to_string(&mut string).await?;
    Ok(string)
}
```

Recursive async functions

What if we want to define an async function recursively? Could we run into problems?

Forgive the change in colorscheme…

```
async fn fibonacci(n: i32) -> i32 {
    match n {
        0 => 0,
        1 | 2 => 1,
        n => fibonacci(n - 1).await + fibonacci(n - 2).await,
    }
}
```

Recursive async functions

What if we want to define an async function recursively? Could we run into problems?

To understand, we need to consider how async functions are implemented…

Forgive the change in colorscheme…

```
async fn fibonacci(n: i32) -> i32 {
    match n {
        0 => 0,
        1 | 2 => 1,
        n => fibonacci(n - 1).await + fibonacci(n - 2).await,
    }
}
```

How is async fn implemented?

Unlike a regular function, an async function doesn't necessarily run to completion.

Every time we write .await, we are creating a **suspend point** in the function. What *actually* happens is that after the first .await, the function returns the `Future` object immediately.

That future object needs to "remember" the execution state of the function (local variables, etc.) so it can **resume** when it gets polled.

We can think of it like a state machine (an enum).

```
async fn async_example(input: String) -> String {
    println!("entering async function...");
    let intermediate_1 = do_work(input).await;
    let intermediate_2 = do_work(input).await;
    let intermediate_3 = do_work(input).await;
    println!("leaving async function...");
}
```

How is async fn implemented?

Unlike a regular function, an async function doesn't necessarily run to completion.

Every time we write .await, we are creating a **suspend point** in the function. What *actually* happens is that after the first .await, the function returns the `Future` object immediately.

That future object needs to "remember" the execution state of the function (local variables, etc.) so it can **resume** when it gets polled.

We can think of it like a state machine (an enum).

```
async fn async_example(input: String) -> String {
    println!("entering async function...");
    let intermediate_1 = do_work(input).await;
    let intermediate_2 = do_work(input).await;
    let intermediate_3 = do_work(input).await;
    println!("leaving async function...");
}
```

Recursive async functions

What if we want to define an async function recursively? Could we run into problems?

```
Diagnostics:
recursion in an async fn requires boxing
a recursive `async fn` call must introduce indirection such as `Box::pin` to avoid an infinitely sized
future [E0733]
```

```
async fn fibonacci(n: i32) -> i32 {
    match n {
        0 => 0,
        1 | 2 => 1,
        n => fibonacci(n - 1).await + fibonacci(n - 2).await,
    }
}
```

Recursive async functions

What if we want to define an async function recursively? Could we run into problems?

Fixed by using `Box::pin` (since Rust 1.77)

```
async fn fibonacci(n: i32) -> i32 {
    match n {
        0 => 0,
        1 | 2 => 1,
        n => Box::pin(fibonacci(n - 1)).await + Box::pin(fibonacci(n - 2)).await,
    }
}
```

Recursive async functions

What if we want to define an async function recursively? Could we run into problems?

Fixed by using `Box::pin` (since Rust 1.77)

```
async fn fibonacci(n: i32) -> i32 {
    match n {
        0 => 0,
        1 | 2 => 1,
        n => Box::pin(fibonacci(n - 1)).await + Box::pin(fibonacci(n - 2)).await,
    }
}
```

Pinning

If we look at the real definition of the `Future` trait, we can see that the poll method doesn't take as argument `&mut self`, but rather `Pin<&mut Self>` (note the `Context` just references the `Waker`).

But what is `Pin`? Long story short, `Pin<T>` guarantees that `T` does not move in memory. This is *stronger* than the guarantee that references make, because we prevent moving operations such as `mem::swap`. This is necessary for self-referential data structures, these are called "position-dependent".

To do this, we define the convention that `T` is a *pointer* to some value (as opposed to the value itself), and we prevent the user from directly manipulating that pointer.

```
pub trait Future {
    type Output;

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

Why do we need `Pin`?

In general, we cannot allow `Future` state machines to be moved in memory. This is because `async` code can contain references.

Consider the example below. We need the ability to store the stack variable `rx` in our state machine, which points to data in `x`. If we “move” our state machine, then the `rx` pointer points to invalid memory.

Most `Futures` don’t require this (namely ones that don’t reference themselves, hence “in general”), so there is a trait called Unpin which lets us access the underlying `T` from a `Pin<T>`. For example `i32` would implement `Unpin`, since it’s not position-dependent.

```
async fn lifetimes() -> i32 {
    let x = 5;
    let rx = &x;

    tokio::time::sleep(Duration::from_secs(2)).await;

    let y = *rx;
    y
}
```

Recursive async functions

What if we want to define an async function recursively? Could we run into problems?

Fixed by using `Box::pin` (since Rust 1.77)

Why does this work?

- Memory for the recursive call gets allocated on the heap
- Its memory location is stable, so we can safely take internal references in the recursive call
- `Box::pin` conveniently creates a value of type `Pin<Box<T>>`

```
async fn fibonacci(n: i32) -> i32 {
    match n {
        0 => 0,
        1 | 2 => 1,
        n => Box::pin(fibonacci(n - 1)).await + Box::pin(fibonacci(n - 2)).await,
    }
}
```

Semantic Properties

What does .await *mean* for our code?

In general, we can rely on the values of our local variables to remain stable across time.

Why?—Rust's borrowing rules guarantee **single ownership** and **exclusive mutable access**. So it's impossible for some other thread to overwrite the values of our local variables (unless we use interior mutability)

```rust
fn main() {
    let x = 10;
    let y = 20;

    do_some_stuff();

    let sum = x + y;
    println!("x + y = {sum}");
}
```

Important Realization:

In the absence of any sharing between threads (via `Mutex`, `RwLock`, channels, etc.), code that uses `.await` will always produce the same result as the equivalent "synchronous" code.

That is, assuming we also don't use `try_join!` or `select!`.

Introducing nondeterminism in a controlled way

Most real-world code relies on some form of nondeterminism (often times this is desirable). This is done in Rust by sprinkling in “interior mutability” types as we’ve discussed in prior lectures.

In **single-threaded** code, we have the property that shared mutable state remains **unchanged** until we run `.await`. Essentially, all the code in between `.await` calls implicitly forms a critical section. (although the compiler still makes us use a `Mutex` unless we are willing to use `unsafe`)

In **multi-threaded** code we have no such guarantee, since any `async` task can concurrently access shared data. `async/await` syntax generally assumes that the code can be run in a multi-threaded context, even if it might not be in practice. This is done to maintain safety and flexibility.

```
async fn task() {
    // initialize to some value
    let shared = Arc::new(Mutex::new(SharedData::default()));

    println!("old value: {}", shared.lock().unwrap().value);
    run_task(Arc::clone(&shared)).await;

    // value might have changed across the `.await` boundary
    println!("new value: {}", shared.lock().unwrap().value);
}
```

Only pay for what you use

Due to these limitations, Rust's `async/await` is very similar in effect to OS-level threading, just with nicer syntax, plus the ability to wait on multiple tasks concurrently (`try_join!` , `select!`).

In principle, *all* code could be implicitly marked `async`, and it would behave the same, solving the function coloring problem. Rust can't do that however, because `async` carries a non-negligible performance cost. This is the kind of tradeoff that a systems language like Rust has to make to be competitive with C/C++ on performance.

So when to use `async`?—It's generally more of a project-wide decision. Mixing `async/await` with synchronous code is awful in practice (I've done it). But generally `async` is great for:

- Web servers
- IO-bound applications
- Highly parallel applications

Also consider the extent to which the libraries you're using support and/or require the use of `async`.

If you're in a multithreaded context and the performance cost of OS-level threads is acceptable, they can be much simpler to use and require less runtime support (since the kernel is responsible for "remembering" the execution state of threads).