

explain me how to desugar `async fn` rust syntax work

Okay, let's dive into the desugaring process of `async fn` in Rust. Desugaring means taking the syntactic sugar (the easy-to-read, high-level syntax) and transforming it into the equivalent, more explicit, lower-level code that the compiler actually understands.

Understanding the `async fn` Sugar

The core idea behind `async fn` is to make asynchronous programming more ergonomic. Instead of manually creating `Futures`, you can write code that looks sequential and synchronous, and the compiler handles the complexity of transforming it into a state machine that can be executed asynchronously.

The Desugaring Steps

Here's a simplified breakdown of how `async fn` desugars:

- 1. State Machine Generation:** The `async fn`'s body is transformed into a state machine. This state machine represents the different points in the function where it might need to suspend execution (e.g., at an `.await` call) and the code to execute when it resumes.
- 2. Future Implementation:** A new type is generated that implements the `Future` trait. This type encapsulates the state machine, the function's local variables, and any captured environment variables.
- 3. poll Method:** The `poll` method of the `Future` implementation contains the logic to drive the state machine forward. Each time `poll` is called, it checks the current state, executes the appropriate code, and potentially transitions to a new state. If the `Future` needs to wait for an asynchronous operation to complete, it returns `Poll::Pending`, indicating that it should be polled again later. Once the `Future` has completed, it returns `Poll::Ready` with the final result.
- 4. Capturing the Environment:** The `async fn` might capture variables from its surrounding environment. These captured variables are stored as fields within the generated `Future` type.
- 5. Returning the Future:** The `async fn` itself is transformed into a regular function that constructs and returns an instance of the generated `Future` type.

Simplified Example

Let's consider a very simple `async fn`:

```
async fn my_async_fn(x: i32) -> i32 {  
    let y = x + 1;  
}
```

```

    y * 2
}

```

Here's a simplified illustration of what the desugared code might look like (note: this is *not* the exact code the compiler generates, but it captures the essence):

```

use std::future::Future;
use std::pin::Pin;
use std::task::{Context, Poll};

// Generated struct to represent the Future
struct MyAsyncFnFuture {
    x: i32,
    y: Option<i32>, // Use Option to track if y has been initialized
    state: MyAsyncFnState,
}

// Enum representing the different states of the state machine
enum MyAsyncFnState {
    Start,
    Finished,
}

impl Future for MyAsyncFnFuture {
    type Output = i32;

    fn poll(mut self: Pin<&mut Self>, cx: &mut Context<'_>) -> Poll<Self::Output> {
        loop {
            match self.state {
                MyAsyncFnState::Start => {
                    let y = self.x + 1;
                    self.y = Some(y);
                    self.state = MyAsyncFnState::Finished; // Move to the next state
                }
                MyAsyncFnState::Finished => {
                    match self.y {
                        Some(y_val) => {
                            let result = y_val * 2;
                            return Poll::Ready(result);
                        },
                        None => panic!("y somehow hasn't been initialised"),
                    }
                }
            }
        }
    }
}

// The original async fn becomes a regular function that returns the Future
fn my_async_fn(x: i32) -> MyAsyncFnFuture {
    MyAsyncFnFuture {
        x,
        y: None,
        state: MyAsyncFnState::Start,
    }
}

```

```

    }
}

#[tokio::main]
async fn main() {
    let result = my_async_fn(5).await;
    println!("Result: {}", result);
}

```

Explanation:

- **MyAsyncFnFuture:** This struct holds the state of the asynchronous computation. It includes the input argument `x`, an optional variable to hold intermediate calculation results, and the current state of the state machine, that represents the async state of the task.
- **MyAsyncFnState:** This enum defines the different states the state machine can be in. In this simple example, it has `Start` and `Finished` states.
- **Future::poll:** This method drives the state machine forward. It checks the current state, performs the corresponding computation, updates the state, and returns `Poll::Pending` if the computation is not yet complete or `Poll::Ready` if it is. The returned type is `Poll<i32>`, because the `async fn` is declared to return type `i32`.
- **my_async_fn (Regular Function):** The original `async fn` is transformed into a regular function that creates and returns an instance of the `MyAsyncFnFuture` struct. You can `.await` on a call to this method.

Example with `.await`

Now let's look at a slightly more complex example with an `.await` call:

```

use tokio::time::{sleep, Duration};

async fn my_async_fn(x: i32) -> i32 {
    sleep(Duration::from_millis(100)).await; // Simulate an asynchronous operation
    let y = x + 1;
    y * 2
}

```

Desugared (Simplified):

```

use std::future::Future;
use std::pin::Pin;
use std::task::{Context, Poll};
use tokio::time::{sleep, Duration};

enum MyAsyncFnState {
    Start,
    AwaitingSleep,
    Finished,
}

```

```

struct MyAsyncFnFuture {
    x: i32,
    y: Option<i32>,
    state: MyAsyncFnState,
    sleep_future: Option<tokio::time::Sleep>,
}

impl Future for MyAsyncFnFuture {
    type Output = i32;

    fn poll(mut self: Pin<&mut Self>, cx: &mut Context<'_>) -> Poll<Self::Output> {
        loop {
            match self.state {
                MyAsyncFnState::Start => {
                    // Initialize the sleep future
                    let sleep_future = sleep(Duration::from_millis(100));
                    self.sleep_future = Some(sleep_future);
                    self.state = MyAsyncFnState::AwaitingSleep;
                    // fallthrough to AwaitingSleep on first poll
                }
                MyAsyncFnState::AwaitingSleep => {
                    // Poll the sleep future
                    let mut sleep_future = self.sleep_future.take().unwrap();
                    match Pin::new(&mut sleep_future).poll(cx) {
                        Poll::Ready(_) => {
                            // Sleep future completed
                            self.state = MyAsyncFnState::Finished;
                            self.sleep_future = None;
                        }
                        Poll::Pending => {
                            // Sleep future is still pending, save it and return Pending
                            self.sleep_future = Some(sleep_future);
                            return Poll::Pending;
                        }
                    }
                }
                MyAsyncFnState::Finished => {
                    let y = self.x + 1;
                    let result = y * 2;
                    return Poll::Ready(result);
                }
            }
        }
    }
}

fn my_async_fn(x: i32) -> MyAsyncFnFuture {
    MyAsyncFnFuture {
        x,
        y: None,
        state: MyAsyncFnState::Start,
        sleep_future: None,
    }
}

#[tokio::main]

```

```
async fn main() {
    let result = my_async_fn(5).await;
    println!("Result: {}", result);
}
```

Key Changes:

- **AwaitingSleep State:** A new state `AwaitingSleep` is added to represent the time when the `Future` is waiting for the `sleep` future to complete.
- **sleep_future Field:** The `MyAsyncFnFuture` struct now has a field `sleep_future` of type `Option<tokio::time::Sleep>` to store the sleep future.
- **Polling the sleep Future:** In the `poll` method, when the state is `AwaitingSleep`, the `sleep` future is polled. If it returns `Poll::Ready`, the state transitions to `Finished`. If it returns `Poll::Pending`, the `sleep_future` is saved back to `self.sleep_future` and `Poll::Pending` is returned, indicating that the `MyAsyncFnFuture` should be polled again later.

Key Concepts and Implications

- **Zero-Cost Abstraction (Mostly):** The goal of `async fn` is to be a zero-cost abstraction. In theory, the desugared code should be as efficient as hand-written state machines. However, in practice, there can be some overhead due to the complexity of the state machine and the need to capture the environment.
- **Stack vs. Heap Allocation:** By default, the `Future` generated by `async fn` is stack-allocated. This is generally more efficient than heap allocation, but it can lead to stack overflow if the `Future` is very large (e.g., if it captures a lot of data). This is one reason why `async_trait` uses boxing (heap allocation) by default, to avoid potential stack overflows.
- **Borrowing and Lifetimes:** The desugared code needs to carefully manage borrowing and lifetimes to ensure that captured variables remain valid across `.await` points. This is why Rust's borrow checker is so important for asynchronous programming.
- **Cancellation Safety:** Rust's `async/.await` is designed to be cancellation-safe. If a `Future` is dropped before it completes, it should not leak resources or cause other problems.

Real-World Complexity

The examples above are highly simplified. In reality, the desugared code can be much more complex, especially when dealing with:

- **Loops and Conditional Statements:** The state machine needs to handle branching and looping logic.
- **Error Handling:** The state machine needs to track whether an error has occurred and propagate it appropriately.
- **Complex Data Structures:** Capturing complex data structures can lead to larger `Future` types and more complex state management.

How to Inspect Desugared Code

While you can't directly see the exact desugared code that the compiler generates (without diving into the compiler's internals), you can get a sense of what's happening by:

- **Reading the Rust Async Book:** The Rust Async Book (<https://rust-lang.github.io/async-book/>) provides a detailed explanation of how `async/.await` works.
- **Using Compiler Explorer:** You can use Compiler Explorer (<https://godbolt.org/>) to examine the generated assembly code. While this won't show you the exact Rust code, it can give you insights into how the compiler is optimizing your asynchronous code.
- **Experimenting with Simple Examples:** By writing simple `async fn` and trying to desugar them manually, you can gain a better understanding of the underlying concepts.

Implications for Kernel/HPC

Understanding the desugaring process can help you write more efficient and robust asynchronous code in your kernel and HPC projects. By being aware of how `async fn` is transformed into a state machine, you can:

- **Minimize Allocations:** Avoid capturing large amounts of data in your `async fn` to reduce the size of the generated `Future` type and minimize the risk of stack overflows.
- **Optimize `poll` Logic:** Write efficient `poll` methods that avoid unnecessary computations or allocations.
- **Ensure Cancellation Safety:** Make sure that your `async fns` are cancellation-safe by properly handling resource cleanup and error propagation.

While the complexity of `async fn` desugaring might seem daunting, it's important to remember that Rust's `async/.await` is designed to be a safe and efficient way to write asynchronous code. By understanding the underlying mechanisms, you can leverage this powerful tool to build high-performance and reliable systems.