# YOSHUA WUYTS

# TREE-STRUCTURED CONCURRENCY — 2023-07-01



For a while now I've been trying to find a good way to explain what structured concurrency is, and how it applies to Rust. I've come up with zingers such as: *"Structured concurrency is structured programming as applied to concurrent control-flow primitives"*. But that requires me to start explaining what structured programming is, and suddenly I find myself 2000 words deep into a concept which seems natural to most people writing programs today [1].

Instead I want to try something different. In this post I want to provide you with a practical introduction to structured concurrency. I will do my best to explain what it is, why it's relevant, and how you can start applying it to your rust projects today. Structured concurrency is a lens I use in almost all of my reasoning about async Rust, and I think it might help others too. So let's dive in.

*This post assumes some familiarity with async Rust and async cancellation. If you aren't already, it might be helpful to skim through the earlier posts on the topic.*

## WHAT IS STRUCTURED CONCURRENCY?

Structured concurrency is a property of your program. It's not just any structure, the structure of the program is guaranteed to be a tree regardless of how much concurrency is going on internally [2]. A good way to think about it is that if you could plot the live call-graph of your program as a series of relationships it would neatly form a tree. No cycles [3]. No dangling nodes. Just a single tree.

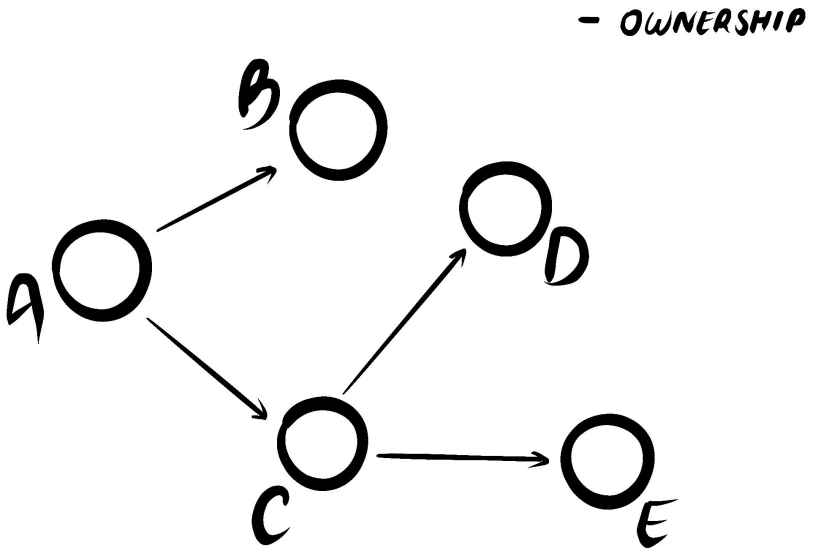


*Fig 1. The arrows point from parent nodes to child nodes. It has no cycles. A parent can have multiple children. But a child always has a single parent - except for the root node.*

And this structure, at least in async Rust, provides three key properties:

- **Cancellation propagation:** When you drop a future to cancel it, it's guaranteed that all futures underneath it are also cancelled.
- **Error propagation:** When an error is created somewhere down in the call-graph, it can always be propagated up to the callers until there is a caller who is ready to handle it.
- **Ordering of operations:** When a function returns, you know it is done doing work. No surprises that things are still happening long after you thought the function had completed.

These properties put together lead to something called a **"black box model of execution"**: under a structured model of computing you don't need to know anything about the inner workings of the functions you're calling, because their behavior is guaranteed. A function will return when it's done, will cancel all work when you ask it to, and you'll always receive an error if there is something

which needs handling. And as a result code under this model is **composable**.

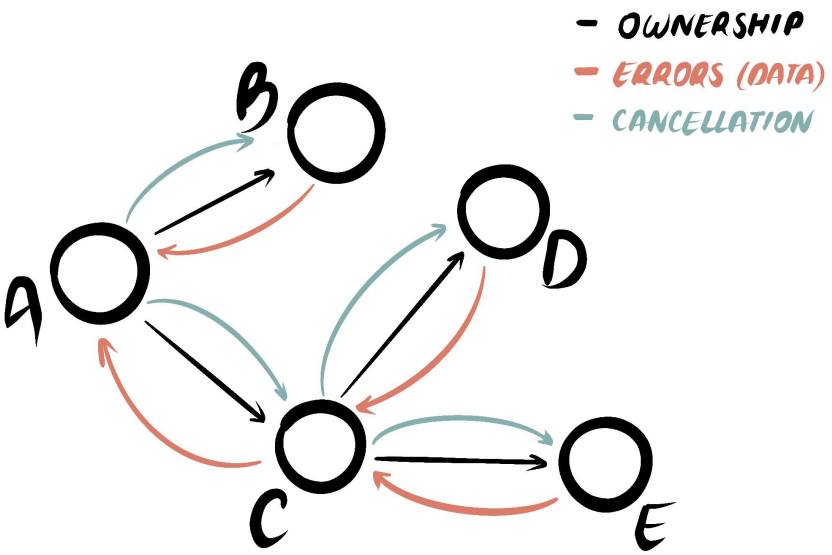


*Fig 2. Under structured concurrency every future has a parent, cancellation flows downward, and errors flow upward. When a future returns, you can be sure it's done doing work.*

If your model of concurrency is *unstructured*, then you don't have these guarantees. So in order to guarantee that say, cancellation is correctly propagated, you'll need to inspect the inner workings of every function you're calling. Code under this model is not composable, and requires manual checks and bespoke solutions. This is both labor-intensive and prone to errors.

## UNSTRUCTURED CONCURRENCY: AN EXAMPLE

Let's start by implementing a classic concurrency pattern: "race". But rather than using *structured* primitives, we can use the staples of unstructured programming: the venerable `task::spawn` and `channel`. The way "race" works is it takes two futures, and we try and get the output of whichever one completes first is whose message we read. We could write it something like this:

```
use async_std::{channel, task};

let (sender0, receiver) = channel::bounded(1);

let sender1 = sender0.clone();
```

```
task::spawn(async move {          // 👆 Task "C"
    task::sleep(Duration::from_millis(100));
    sender1.send("first").await;
});

task::spawn(async move {          // 👆 Task "B"
    task::sleep(Duration::from_millis(100));
    sender0.send("second").await;
});

let msg = receiver.recv().await; // 👆 Future "A"
println!("{msg}");
```

While this implements "race" semantics correctly, it doesn't handle cancellation. If one of the branches completes, we'd ideally like to cancel the other. And if the containing function is cancelled, both computations should be cancelled. Because of how we've structured the program neither task is anchored to a parent future, and so we can't cancel either computation directly. Instead the solution would be to come up with some design using more channels, anchor the handles - or we could instead rewrite this using structured primitives.

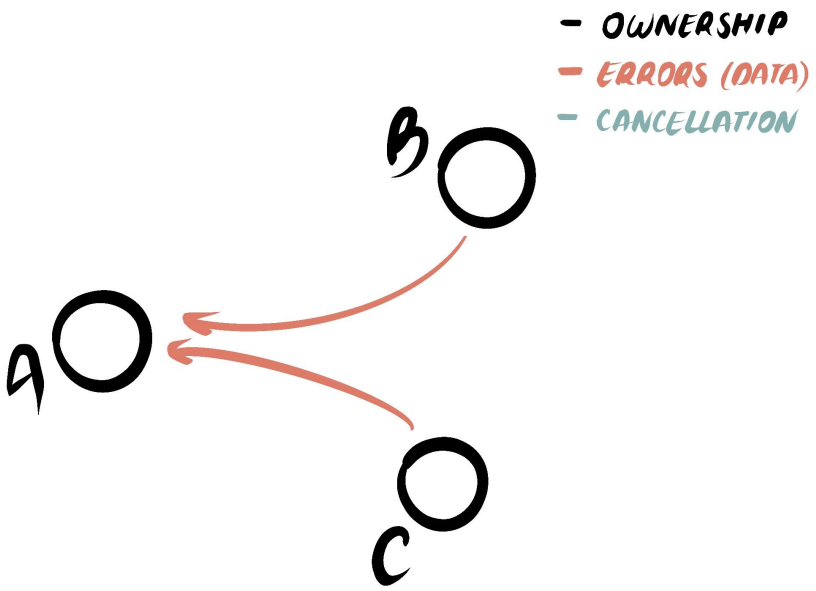


*Fig 3. You can create a "race" operation by combining tasks and channels. Data can flow out of the tasks to the caller. But because the tasks aren't rooted in a parent task, cancellation doesn't propagate.*

## STRUCTURED CONCURRENCY: AN

# EXAMPLE

We can rewrite the example above using structured primitives instead. Rather than DIY-ing our own "race" implementation using tasks and channels, we should instead be using a "race" primitive which implements those semantics for us - and correctly handles cancellation. Using the futures-concurrency library we could do that as follows:

```
use futures_concurrency::prelude::*;
use async_std::task;

let c = async {                    // 👆 Future "C"
    task::sleep(Duration::from_millis(100));
    "first"
};

let b = async {                    // 👆 Future "B"
    task::sleep(Duration::from_millis(100));
    "second"
};

let msg = (c, b).race().await;     // 👆 Future "A"
println!("{msg}");
```

When one future completes here, the other future is cancelled. And should the `Race` future be dropped, then both futures are cancelled. Both futures have a parent future when executing. Cancellation propagates downwards. And while there are no errors in this example, if we were working with fallible operations then early returns would cause the future to complete early - and errors would be handled as expected.

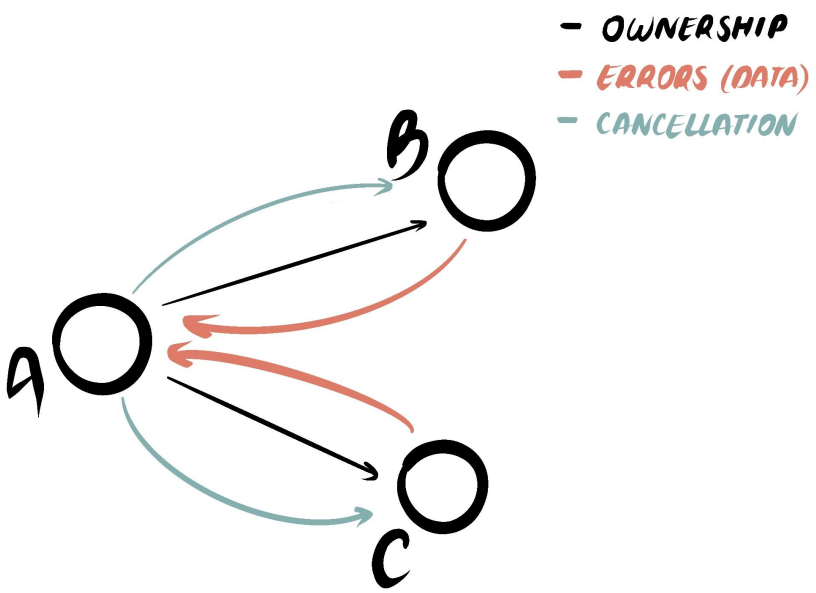


*Fig 4. By using a structured "race" primitive all child futures are rooted in a parent future. Which allows both cancellation and errors to propagate. And the operation won't return until all child futures have dropped.*

So far we've looked at just the "race" operation, which encodes: *"Wait for the first future to complete, then cancel the other"*. But other async concurrency operations exist as well, such as:

- **join**: wait for all futures to complete.
- **race_ok**: wait for the first future to complete which returns 0k .
- **try_join**: wait for all futures to complete, or return early if there is an error.
- **merge**: wait for all futures to complete, and yield items from a stream as soon as they're ready.

There are a few more such as "zip", "unzip", and "chain" - as well as dynamic concurrency primitives such as "task group", "fallible task group", and more. The point is that the set of concurrency *primitives* is bounded. But they can be recombined in ways that makes it possible express any form of concurrency you want. Not unlike how if a programming language supports branching, loops, and function calls you can encode just about any control-flow logic you want,- without ever needing to use "goto".

## WHAT'S THE WORST THAT CAN HAPPEN?

People sometimes ask: What's the worst that can happen when you

don't have structured concurrency? There are a number of bad outcomes possible, including but not limited to: data loss, data corruption, and memory leaks.

While Rust guards against *data races* which fall under the category of "memory safety", Rust can't protect you from logic bugs. For example: if you execute a `write` operation inside of a task whose handle isn't joined, then you'll need to find some alternate mechanism to guarantee the ordering of that operation in relation to the rest of the program. If you get that wrong you might accidentally write to a closed resource and lose data. Or perform an out-of-order write, and accidentally corrupt a resource [4]. These kinds of bugs are not in the same class as memory safety bugs. But they are nonetheless serious, and they can be mitigated through principled API design.

## APPLYING STRUCTURED CONCURRENCY TO YOUR PROGRAMS

**task::spawn**

When using or authoring async APIs in Rust, you should ask yourself the following questions to ensure structured concurrency:

1. **Cancellation propagation**: If this future or function is dropped, will cancellation propagate to all child futures?
2. **Error propagation**: If an error happens anywhere in this future, can we either handle it directly or surface it to the caller?
3. **Ordering of operations**: When this function returns, will no more work continue to happen in the background?

If all of these properties are true, then once the function exits it's done executing and you're good. This however leads us to a major issue in today's async ecosystem: neither async-std nor tokio provide a `spawn` function which is structured. If you drop a task handle the task isn't cancelled, but instead it's detached and will continue to run in the background. This means that cancellation doesn't automatically propagate across task boundaries, causing it to be unstructured.

The smol library gets closer though. It has a task implementation which gets us closer to "cancel on drop"-semantics out of the box.

Though it doesn't get us all the way yet because it doesn't guarantee an ordering of operations. When a smol `Task` is dropped the task isn't guaranteed to have been cancelled, all it guarantees is that the task will be cancelled at some point in the future.

**async drop**

Which brings us to the biggest piece missing from async Rust's structured concurrency story: the lack of async Drop in the language. Smol's tasks have an async cancel method which only resolves once the task has successfully been cancelled. Ideally we could call this method in the destructor and wait for it. But in order to do that today we'd need to block the thread, and that can lead to throughput issues. No, in practice what we really need for this to work well is async destructors [5].

**what can you do today?**

But while we can't yet trivially fulfill all requirements for async structured concurrency for async tasks, not all hope is lost. Without async Drop we can already achieve 2/3 of the requirements for task spawning today. And if you're using a runtime other than smol, adapting the spawn method to work like smol's does is not too much work. But most concurrency doesn't need tasks because it isn't dynamic. For that you can take a look at the futures-concurrency library which implements composable primitives for structured concurrency.

If you want to adopt structured concurrency in your codebase today, you can start by adopting it for non-task-based concurrency. And for task-based concurrency you can adopt the smol model of task spawning to benefit from most of the benefits of structured concurrency today. And eventually the hope is we can add some form of async Drop to the language to close out the remaining holes.

## PATTERN: MANAGED BACKGROUND TASKS

People frequently ask how they can implement "background tasks" under structured concurrency. This is used in scenarios such as an HTTP request handler which also wants to submit a piece of telemetry. Rather than blocking sending the response on the telemetry, it spawns a "background task" to submit the telemetry in

the background, and immediately returns from the request. This can look something like this:

```
let mut app = tide::new();
app.at("/").post(|_| async move {
    task::spawn(async {  // 👈 Spawns a background task…
        let _res = send_telemetry(data, more_data).await;
        // … what if `res` is an `Err`? How should we handl
    });
    Ok("hello world")   // 👈 …and returns immediately afte
});
app.listen("127.0.0.1:8080").await?;
```

The phrase "background task" seems polite and unobtrusive. But from a structured perspective it represents a computation without a parent - it is a *dangling task*. The core of the pattern we're dealing with is that we want to create a computation which outlives the lifetime of the request handler. We can resolve this by rather than creating a dangling task to submit it to a task queue or task group which outlives the request handler. Unlike a dangling task, a *task queue* or *task group* preserves structured concurrency. Where a dangling task doesn't have a parent future and becomes unreachable, using a task queue we transmit the ownership of a future to a different object which outlives the current more ephemeral scope.

I've heard people make the argument before that `task::spawn` is perfectly structured, as long as you think of it as spawning on some sort of unreachable, global task pool. But the question shouldn't be whether tasks are spawned on a task pool, but what the relationship is of those tasks to the rest of the program. Because we cannot cancel and recreate an unreachable task pool. Nor can we receive errors from this pool, or wait for all tasks in it to complete. It doesn't provide the properties we want from structured concurrency - so we shouldn't consider it structured.

I don't feel like the ecosystem has any great solutions to this yet - in part limited because we want "scoped tasks" which basically require linear destructors to function. But other experiments exist so we can use that plus channels to put something together which gives us what we want:

⚠️ *Note: This code is not considered "good" by the author, and is*

*merely used as an example to show that this is possible to write today. More design work is necessary to make this ergonomic ⚠️*

```
// Create a channel to send and receive futures over.
let (sender, receiver) = async_channel::unbounded();

// Create a structured task group at the top-level, next to
//
// If any errors are returned by the spawned tasks, all act
// and the error is returned by the handle.
let telemetry_handle = async_task_group::group(|group| asyn
    while let Some(telemetry_future) = receiver.next().awai
        group.spawn(async move {
            telemetry_future.await?;  // 👈 Propagate error
            Ok(())
        });
    }
    Ok(group)
});

// Create an application state for our HTTP server
#[derive(Clone)]
struct State {
    sender: async_channel::Sender<impl Future<Result<_>>>,
}

// Create the HTTP server
let mut app = tide::new();
app.at("/").post(|req: Request<State>| async move {
    state.sender.send(async {   // 👈 Sends a future to the
        send_telemetry(data, more_data).await?;
        Ok(())
    }).await;
    Ok("hello world")           // 👈 …and returns immediat
});

// Concurrently execute both the HTTP server and the teleme
// and if either one stops working the other stops too.
(app.listen("127.0.0.1:8080"), telemetry_handle).race().awa
```

Like I said: we need to do a lot more API work to be able to rival the convenience of just firing off a dangling task. But what we lack for in API convenience, we make up for in semantics. Unlike our earlier example this will correctly propagates cancellation and errors, and every executing future is owned by a parent future. We could even

take things a step further and implement things like retry-handlers with error quotas on top of this to create a more resilient system. But hopefully this is enough already to get the idea across of what we could be doing with this.

## GUARANTEEING STRUCTURE

I've been asking myself for a while now: *"Would it be possible for Rust to enforce structured concurrency in the language and libraries?"* I don't believe this is something we guarantee from the language. But it is something *can* guarantee for Rust's library code, and make it so most async code is structured by default.

The reason why I don't believe it's fundamentally possible to guarantee structure at the language level is because it's possible to express any kind of program in Rust, which includes unstructured programs. Futures, channels, and tasks as they exist today are all just regular library types. If we wanted to enforce structure from the language, we would need to find a way to disallow the creation of these libraries - and that seems impossible for a general-purpose language [6].

Instead it seems more practical to me to adopt tree-structured concurrency as the model we follow for async Rust. Not as a memory-safety guarantee, but as a design discipline we apply across all of async Rust. APIs which are unstructured should not be added to the stdlib. And our tooling should be aware that unstructured code may exist, so it can flag it when it encounters it.

## CONCLUSION

In this post I've shown what (tree-)structured concurrency is, why it's important for correctness, and how you can apply it in your programs. I hope that by defining structured concurrency in terms of guarantees about propagation of errors and cancellation, we can create a practical model for people to reason about async Rust with.

As recently reported by Google, async Rust is one of the most difficult aspects of Rust to learn. It seems likely that the lack of structure in async Rust code today did not help. In async code today neither cancellation nor errors are guaranteed to propagate.

This means that if you want to reliably compose code, you need to have knowledge of the inner workings of the code you're using. By adopting a (tree-)structured model of concurrency these properties can instead be guaranteed from the outset, which in turn would make Async Rust easier to reason about and teach. Because *"If it compiles it works"* should apply to async Rust too.

*Thanks to Iryna Shestak for illustrating and proof-reading this post.*

## NOTES

**1.** If you're interested in structured programming though, I can recommend Dijkstra's writing on the subject. Most programming languages we use today are structured, so reading about a time when when that wasn't the case is really interesting. ←
**2.** When I was researching this topic last year I found a paper from I believe the 80's which used the phrase "tree-structured concurrency". I couldn't find it again in time for this post, but I remember tweeting about it because I hadn't seen the term before and I thought it was really helpful! ←
**3.** Yes yes, recursion can make functions call themselves - or even call themselves through a proxy. By "live call-graph" I mean it like how flame charts visualize function calls. Recursion is visualized by stacking function calls on top of each other. The same idea would apply here by adding new async nodes as children of existing nodes. The emphasis here is very much on *live* call-graph, not *logical* call-graph. ←
**4.** At a previous job we experienced exactly this in a database client: we were having issues propagating cancellation correctly, which meant that the connection protocol could be corrupted because we didn't flush messages when we should have. ←
**5.** Async cancellation is hardly the only motivation for async Drop. It also prevents us from encoding basic things like: "flush this operation on drop" - which is something we *can* encode in non-async Rust today. ←
**6.** An example of this from structured programming: Rust is a structured language. Assembly is not a structured language. You can implement an assembly interpreter entirely in safe Rust - meaning you can express unstructured code in a structured language. I could show examples of this, but eh I hope the general line of reasoning makes sense here. ←

## REFERENCES

▸ View all references