YOSHUA WUYTS ←

AUTOMATIC INTERLEAVING OF HIGH-LEVEL CONCURRENT OPERATIONS — 2025-05-05

1. introduction
2. the problem
3. the solution
4. further reading

INTRODUCTION

When working with low-level concurrency (atomics), programming languages are generally quite eager for compilers to reorder operations if it leads to better performance. Information about whether it's ok to reorder operations is encoded using Atomic Orderings, Fences, and Operations. It's strange that most programming languages that support semantic-aware reordering of low-level concurrent operations, don't also include similar support for reordering the execution of high-level concurrent operations.

To my knowledge this is true for most languages, with the notable exception of Swift and its `async let` construct. This feature preserves the linear-looking nature of async code, but allows the compiler to inspect the control flow graph and schedule operations concurrently where possible. That means that just like with atomics, an operation that is defined later in a piece of code may finish execution before an operation that appears earlier. Here is an example Swift program where everything that *can* be concurrent actually *is* concurrent:

```
func makeDinner() async throws -> Meal {
  async let veggies = chopVegetables()                    /
  async let tofu = marinateTofu()                         /
  async let oven = preheatOven(temperature: 350)          /

  let dish = Dish(ingredients: await [try veggies, tofu]) /
  return await oven.cook(dish, duration: .hours(3))       /
}
```

THE PROBLEM

To me this represents the pinnacle of language-level support for asynchronous/concurrent programming. It makes it trivial to change any code that *may* be run concurrently to actually run concurrently. It enables the compiler to take care of what is otherwise tedious and/or illegible. Take for example this code that's written in a serial fashion using async/.await:

```
async fn make_dinner() -> SomeResult<Meal> {
    let veggies = chop_vegetables().await?;
    let tofu = marinate_tofu().await?;
    let oven = preheat_oven(350).await;

    let dish = Dish(&[veggies, tofu]).await;
    oven.cook(dish, Duration::from_mins(3 * 60)).await
}
```

Using `Future::join` operations we can rewrite it to execute independent operations concurrently. But this comes with the downside that the code is now significantly less legible. Here is the same code, written using `Future::try_join`:

```
use futures_concurrency::prelude::*;

async fn make_dinner() -> SomeResult<Meal> {
    let dish_fut = {
        let veggies_fut = chop_vegetables();
        let tofu_fut = marinate_tofu();
        let (veggies, tofu) = (veggies_fut, tofu_fut).try_j
        Dish::new(&[veggies, tofu]).await
    };
    let oven_fut = preheat_oven(350);
    let (dish, oven) = (dish_fut, oven_fut).try_join().awai
    oven.cook(dish, Duration::from_mins(3 * 60)).await
}
```

To capitalize on one of the core features of `async/.await` (*ad-hoc concurrent scheduling*), we had to sacrifice one of its main benefits (legibility). It's not good when two core parts of the same feature are in tension with each other like that. And we can't just wave a wand and tell the compiler to automatically execute these futures concurrently. Futures tend to express operations that change program state in one way or another. That is to say: most futures encode *side-effects*. And the compiler can't automatically infer which side effects can be executed serially and which can be executed concurrently. That's because it's not aware of the program *semantics*.

THE SOLUTION

The solution is to allow programmers to provide opt-in to explicit reorderings in their code, just like Swift does using `async let`. We could use a concise notation along the lines of `.co.await` (this is a strawman, pick your own favorite notation). We want the notation to be in postfix position because unlike Swift we don't want to eagerly start executing when operations are defined, but only affect the way operations are scheduled when `.await`ed. And this way we never have to actually have to represent it in the type system either [1]. This would look something like this:

```
async fn make_dinner() -> SomeResult<Meal> {
    let veggies = chop_vegetables().co.await?;
    let tofu = marinate_tofu().co.await?;
    let oven = preheat_oven(350).co.await;

    let dish = Dish(&[veggies, tofu]).co.await;
    oven.cook(dish, Duration::from_mins(3 * 60)).await
}
```

This code would directly lower to the equivalent of the `Future::join`-based scheme. But with the benefit of needing far less ceremony to encode the same semantics. The other benefit of this scheme is that we always retain the option to schedule these operations serially if we choose to. That makes this scheme compatible with async-polymorphic functions, where manual `Future::join` calls are not.

A feature along these lines is important, because in order to make full use of async Rust's concurrent scheduling capabilities, any operations which can be executed concurrently should be executed concurrently. But without language support that comes at a severe cost to legibility, and in turn maintainability. The only way out of this dilemma is to do what Swift has done and directly include language support for it.

FURTHER READING

- Tasks are the Wrong Abstraction - Introduces the `ParallelFuture` trait, which can be combined with the scheme outlined in this post to automatically schedule futures on multiple cores.
- Tree-Structured Concurrency - Discusses what structured concurrency is, how to reason about it, and what's missing for Rust to fully support it.
- Extending Rust's Effect System - Discusses among other things async-polymorphic functions.

NOTES

1. Writing `.co` without following it with an `.await` should be a compiler error. The `.co` would serve as a modifier on the `.await`. Though perhaps something like C++'s `co_await` notation is simpler. Whatever the syntax though, I don't think it should ever surface to the type system. ←

REFERENCES

▸ View all references