

Applicatives, Monads, and a Future for Rust

Mike He

Jiuru Li

Yihong Zhang

Background

Concurrency is now ubiquitous in today’s network programming. In a typical scenario, an application makes requests to remote servers and program logic will depend on the collected responses. However, the traditional sequential execution model will waste CPU time by waiting for the responses. Therefore, concurrency is introduced to make sure the part of programs not dependent on the server response can make progress and maximize execution efficiency. Still, how to maximize concurrency without burdens on the developers is an interesting programming language design problem.

Async is a concurrent programming model adopted in Rust and has now been widely used. To use the async model, developers create an encapsulated async block, called future, for each remote data fetch, and let an executor run many futures concurrently. Compared to the multi-processing or multi-threading concurrency model, async does not bear system call overheads and is more lightweight and more efficient. However, there are several problems with the async programming model, potentially preventing it from further adoption compared to multi-threading. For example, API calls need to be asynchronous in order to be run concurrently. Moreover, the executor can only execute multiple futures together when they are explicitly used as parameters to specific future calls such as `join!` or `select!`.

In this project, we propose an alternative concurrency model for Rust based on Haxl, a concurrency framework developed by Facebook. Our programming model allows mixing asynchronous and synchronous calls in the same program and schedule concurrent tasks automatically without explicit calls to future APIs like `join!` or `select!`. At the core of our programming model is applicative functors, or applicatives, and monads. In fact, monads have been long known in the functional programming community for being able to express concurrency, and the async model in Rust can be seen as syntactic sugars around the concurrency monad. However, following Haxl, we propose to use applicative functors to express tasks that do not have dependencies and can be executed in parallel. The use of applicatives allows

us to batch independent tasks via standard applicative operations like `bind` and `ap`. However, this will require the user to manually call these applicative operations, causing mental burdens. The async model in Rust eases this problem by adding new syntactic constructs like `await` and `async`, which is not practical for us. Therefore, we propose to use Rust’s macro mechanism to mitigate the syntactic indirections as much as possible.

Design of Ruxl

Fetch<T>. We use trait `Request<T>` to represent an atomic async request of type T, and use `Fetch<T>` to represent a program of type T that could possibly make several requests. First, any plain Rust value of type T and any struct implementing `Request<T>` can be lifted into a `Fetch<T>`:

```
fn pure(a: T) -> Fetch<T>
fn new<R: Request<T>>(request: R) -> Fetch<T>
```

These are the “leaves” of a `Fetch<T>` program.

Moreover, `Fetch<T>` forms a monad, because the following operation can be implemented for `Fetch<T>`:

```
fn bind<T, U>(x: Fetch<T>,
               k: impl FnOnce(T) -> Fetch<U>)
    -> Fetch<U>
```

This essentially says that, give me a program of type `Fetch<U>` that is dependent on the choice of value T and give me a program of type `Fetch<T>`, I can tell you how to compose both programs and give you a program of type `Fetch<U>` without any dependency. This intuitively explains why monad is a good abstraction for sequential execution.

Finally, `Fetch<T>` is also an applicative functor:

```
fn ap<T, U>(
    f: Fetch<impl FnOnce(T) -> U>,
    x: Fetch<T>)
    -> Fetch<U>
```

`ap` takes a program that computes a (higher-order) function from T to U and a program that computes to T and produces a program that computes to U. The `ap`

operator describes the nature of parallel computation. This can be more clearly demonstrated by a variant of it, `ap2`, which can be implemented using `ap`:

```
fn ap2<T1, T2, U>(
  f: Fetch<impl FnOnce(T1, T2) -> U>,
  x: Fetch<T1>, y: Fetch<T2>
) -> Fetch<U>
```

`ap2` reads: give me a program that computes a function from `(T1, T2)` to `U` and individual programs that compute to `T1` and `T2`, I can give you a program that computes to `U`. In this case, the implementation of `ap` can freely parallelize the computation of `Fetch<T1>` and `Fetch<T2>` (and `Fetch<impl FnOnce(T1, T2) -> U>`), because there is no inter-dependencies between them.

It turns out these are all we need to express all the features we want: sequential composition, method calls, and other advanced control flow.

An acute reader may notice that all monads are applicatives, and applicatives can be automatically implemented using operations on monads, so why do we even call them out? We do this, following Haxl, because of the restrictions specific to applicatives. In this case, compositions of applicatives are known to be parallelizable, so a more efficient implementation can be derived.