What a better Rust would look like

Wed, Apr 27, 2022

The Hare programming language was announced a few days ago, and, at first glance, its syntax looks similar to Rust.

So, why would people bother to create a new language which aims to fulfill the same niche as Rust (system programming), with a similar syntax?

Rust is often described by its fans (me included) as the most advanced programming language, perfectly mixing Computer Science and real-world usability. Yet, Rust is far from perfect and may often be frustrating for developers.

So today, let's start from the axiom that Rust is the least bad programming language and what it would take to make it the "perfect" language and ecosystem.

Standard library

An anemic standard library with an integrated package manager (`cargo`) leads to a culture of a huge dependency tree of small packages and thus a huuuge attack surface for whoever wants to compromise a project.

Also, it adds burdens on consumers of these dependencies to keep them up to date and update their code from time to time due to API changes.

So in a few words: a small standard library is wasting a lot of time for everyone involved.

In 2022, a standard library should at least contain the following packages:

- crypto
- HTTP
- Serialization / Deserialization: JSON
- Text/HTML Templates

- rand
- Encoding / Decoding: base64, base32, hex...
- logging
- Time / Date handling
- Url handling
- UUID/ULID
- CLI arguments parsing
- Images manipulation
- Filesystem
- Emails
- Regex
- ...

Packages management

I don't like centralized package repositories. They add complexity and obfuscation, all while supply chain attacks are increasing.

It is possible to import packages from Git in Rust, but as it's not the "official" way, the packages you import from Git will surely import packages from crates.io themselves 🤷

So please, follow the Go model: centralized discovery but decentralized distribution. It's not perfect, but it certainly reduces the chances of a successful supply chain attack.

```
[dependencies]
sqlx = { git = "https://github.com/launchbadge/sqlx", version = "0.5.13" }
different_name = { git = "https://github.com/skerkour/something", version = "1"
```

```
use {
  sqlx::Postgres,
  different_name::Something,
}

// ...
```

Modules

Due to how modules and packages work in Rust, I create dependency cycles more often than with some other languages.

Also, other than for tests, I may have used multiple modules per file only once or two! And even for tests, I would prefer a separate file. So Rust's modules system only adds verbosity with `super` & Co.

I think that Go got it right: modules are scoped by folder instead of files.

Visibility and Privacy

Again, I think that Go nailed it: Using the case (lowercase for private, uppercase for public) of the first letter of the identifier is perfect for lazy developers like me.

It stays relatively explicit but less verbose than using `pub` everywhere, as per Rust's explicitness ethos.

```
struct thisIsPrivate {
  private bool
  Public bool
}

struct ThisIsPublic {
  // ...
}
```

Memory management

The borrow checker is really nice and is one of the principal reasons you want to use Rust: catching bugs at compile time.

Yet, I don't think that lexical lifetimes are the answer.

I'm far from an expert in this field, still, from a programmer's perspective, I would love to

see a mix of compile-time lifetime analysis, Automatic Reference Counting (like in the lobster programming language), and manual memory management (marked as `unsafe`) for when extreme performance is needed.

I no longer want to see lifetime annotations pollute our code ever :)

Instead of

```
struct Foo {
  bar: Bar
}

struct Foo<'a> {
  bar: &'a Bar
}

struct Foo<'a> {
  bar: &'a mut Bar
}

struct Foo {
  bar: Box<Bar>
}

struct Foo {
  bar: Rc<Bar>
}

struct Foo {
  bar: Arc<Bar>
}
```

We would just have to:

```
struct Foo {
  bar Bar
}
```

Release schedule

I believe that Rust moves too fast.

A programming language is a **platform** that other people build on top of, not a product. Moving fast is good for products, but bad for platforms because they need to maintain compatibility and thus can't really/easily remove features.

Thus features only accumulate, and complexity compounds over time.

Until the day where when different teams use completely different features of the languages and we can no longer understand the code of other people.

Governance

Related to features bloat: Who is in charge of refusing new features added to the language to avoid its collapse?

Some Closing Thoughts

So yeah... We are no longer talking about Rust, but a completely different language.

Also, is this list objective and exhaustive? Of course not. It's the fruit of my experience working with C, Python, TypeScript, Go, and Rust.

Rust is far from perfect, so do we still need to rewrite everything that is currently in unsafe languages in Rust? **Yes**!

Today Rust already does a lot of things right, and if in the future the *perfect language*™ exists, we will be able to link the existing safe and robust Rust libraries to this new language, via FFI for example.