Towards Impeccable Rust

Rust is now used in critical software.

“Towards” & “Impeccable”

Def.: free from fault or blame.

- Impossible to reach perfection.
  - We should strive to get close!
- Impeccable software can fail.
  - e.g., fault in spec or hardware.

Towards Impeccable Rust

Signs you’re doing it right.

Jon Gjengset / @jonhoo

Principal Engineer @ Helsing
Author Rust for Rustaceans
General Rust educator





Notional image only (generated by stable diffusion)



Notional image only (generated by stable diffusion)



Notional image only (generated by stable diffusion)

Jon Gjengset / @jonhoo

Principal Engineer @ Helsing
Author Rust for Rustaceans
General Rust educator



There is no single trick.

Testing

Be paranoid

it works != it’s not broken

Assert *everything*.

Miri

detects undefined behavior (and leaks) even if nothing panics.

Sanitizers

detect problematic threading and memory access patterns.

Aside: have tests for your error cases!

Embrace chaos

Anything that can go wrong will go wrong (and at the worst possible time).

`turmoil` / `shuttle`
async / sync chaos

`quickcheck` / `proptest`
value chaos

`cargo-mutants`
logic chaos

Be exhaustive

(where you can)

Loom

all possible and distinguishable concurrent executions.

Kani

all possible and distinguishable inputs.

```
if x == 9000 {
  panic!(“one short”);
} else {
  println!(“all is well”);
}
```

Be exhaustive

(where you can)

Loom

all possible and distinguishable concurrent executions.

Kani

all possible and distinguishable inputs.

Benchmarks

Know thyself

Benchmarks should capture the entire performance profile.

That means:

- pathological cases
- micro **and** macro
- under, at, **and** over capacity
- on all relevant targets

Are your benchmarks **useful**?

Have trustworthy measurements

Let your CI fail on regression.

This isn’t easy!

Don’t use time (incl. ops/second):
e.g., iai-callgrind

Run old and new interleaved:
e.g., tango

Minimize noise:
dedicated host + <100% load.

Speed isn’t all

Measure all that matters

Throughput & “Goodput”

Memory use (avg. and max)

Latency

Speed isn’t all

Measure **all** that matters

Outcomes

1. Simulate real-life inputs
2. Measure system outputs
3. Compare output to ground truth

Simple benchmarks lie

a.k.a. “I ran it in a loop and got a bigger number”

How you benchmark matters
open vs closed vs partly-open
representative workloads

What you record matters
mean, median, histogram, cdf

How you compare matters
y > x is **not** sufficient

Documentation

Document decisions taken

Why does it work like this?

In particular:

- What alternatives were discarded and why?
- What downsides were explicitly accepted and why?

No perfect method. Try (Y)ADRs?
make sure there's *something*

Document what's not there

What shortcuts did we take?

Missing handling of corner-cases.

`todo!()` / `unimplemented!()`

Future optimization opportunities.

Absence of an `impl` (like `From`).

Misuse resistance

Make misuse inexpressible

"You're holding it wrong" is not acceptable.

Newtypes (not aliases).
Meters(u64) vs Miles(u64)

Typestates.
Rocket<Ground> vs Rocket<Air>

Two-phase structs.
TomlConfig vs ResolvedConfig

Enums over booleans.
foo(true, true, false, true) 👀

Enums for linked arguments.
f(true, Some(_)) ⊕ f(false, None)

Follow idioms

Surprise and misuse
go hand in hand.

Clippy is your friend.

The Rust API Guidelines \o/

If it smells like OOP or
Python/Java/C, think again.
I wish I could describe the smell…

Compatibility

Minimize hazards

There are more than you think.
Consider a simple & stable core.

Concrete types

prefer `-> impl Trait`

avoid `pub` fields

Public dependencies

args, return types, trait impls, …

`impl From`

prefer non-pub inherent methods!

Humans miss things

Best-effort automation is better than nothing!

Tooling here is desperately needed.

`cargo-semver-checks` is great, but has a long way to go (see #5).

`cargo-public-api` is useful, but mostly about high-level API.

Dependencies

Knowledge is everything

"Which active devices are running libsqlite3-sys < 0.25.1 or sqlite < 3.39.2?"

Healthy skepticism is a given.

Then tracking everything.
the **complete** dependency closure
and every deployment and device

Then connecting to known issues.
e.g., join against RUSTSEC

Then vetting for unknown issues.
e.g., cargo-vet (public or internal)

Make stagnation a recurrent choice

Cost goes up for each “skip”!

Have **loud** reminders when:

- you’re behind
- dependencies are dead

Reduce friction to catching up:

- Auto-merging bump PRs.
- Budget for maintenance work.
- Upstream changes (no forks!).
- Wrap unstable dependencies.

QA

So, (some of) what it takes:

- Test exhaustively for all the things you haven’t thought of.
- Have trustworthy, realistic benchmarks for everything.
- Ensure past knowledge is not lost.
- Make it unlikely for developers to write incorrect code.
- Educate on and manage backwards compatibility.
- Track dependencies thoroughly.
- Make stagnation a recurrent choice.