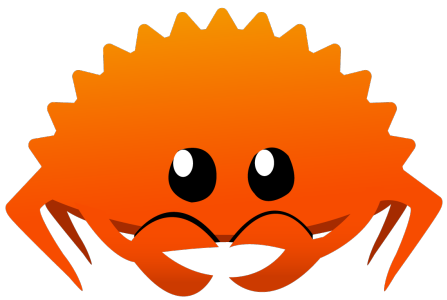
# Concurrency

Lukas Prokop

August 5, 2020

RustGraz community



# Prologue

# Recap: Rusty days


- Webference organized by Rust Wrocław
- 8 talks between 2020-07-27 and 2020-07-31
- I will give a short talk summary

**Mon, 2020-07-27 — Steve Klabnik**

**“Should we have a Rust 2021 edition?”**

# Recap: Rust 2021 edition Talk

rustc is a multi-pass compiler: AST → HIR → MIR → LLVM-IR

**HIR** expand source code to simpler primitive statements (type checking, method lookup)

**MIR** is all about control flow (control-flow, borrow checking)

- *lowering*: “The MIR is then lowered to LLVM-IR”
- “cargo check” omits code generation step
- *goal*: “query-based” compiler (memoized answers)
- editions are not allowed to differ in MIR

**nightly release** every night

**stable + beta release** every six weeks

# Recap: Rust 2021 edition Talk

**Editions:**

| | |
|---|---|
| **is** | longer-term progress, breaking changes, new editions are opt-in, can't change everything |
| **can** | can introduce new keywords, repurpose syntax (e.g. deprecate trait, introduce dyn trait) |
| **cannot** | cannot change coherence rules, standard library |

- when do editions happen? no policy
- edition 2018 (= rust 1.31) was "a really big project for the various teams" and "a lot of burnout amongst contributors" ("I was a total mess")
- editions feature-driven or time-boxed? time-boxed! 3 years is a nice compromise between yearly and 5-year releases
- Steve: We should have a 2021 edition. Much smaller than 2018.

# Recap: Rust 2021 edition Talk

```
meisterluk@gardner ~ % cat test.rs
fn async() -> u64 {
    42
}

fn main() {
    println!("{}", async());
}

meisterluk@gardner ~ % rustc test.rs
```

# Recap: Rusty days

```
meisterluk@gardner ~ % rustc --edition=2018 test.rs
error: expected identifier, found keyword `async`
 --> test.rs:1:4
  |
1 | fn async() -> u64 {
  |    ^^^^^ expected identifier, found keyword
  |
help: you can escape reserved keywords
      to use them as identifiers
  |
1 | fn r#async() -> u64 {
  |    ^^^^^^^
```

**Mon, 2020-07-27 — Michalina Kotwica**

**“Low-level optimization of algebraic and similar structures”**

# Recap: Low-level optimization Talk

- type algebra
- unit & never type
- memory layout (ABI) of composite types (struct, enum, futures)
- struct Bar(u16, u8, u16, u8):
  `C: 11 11 33 xx 22 22 44 xx` `Rust: 11 11 22 22 33 44` (fields reordered)
- `Option` uses a discriminator, `Option<Option<u64>>` uses only one discriminator
- 41.2 % of enums have no type arguments, 16.4 % of enums have one type argument, 23.7 % of enums have two type arguments

**Tue, 2020-07-28 — Peter Parkanyi**

**“Fast encrypted backups with Rust - ‘How I stopped worrying and love mmap’”**

# Recap: Fast encrypted backups Talk

https://github.com/rsdy/zerostash

- End-to-End Encryption
- latency and throughput
- Zero-metadata data stash: deduplicated, works as a file system and key/value store
- Cryptographic primitives:

  **Passwords** Argon2

  **Indexing** Blake2

  **Compression** LZ4

  **Encryption** ChaCha20-Poly1305

  **Deduplication** SeaHash
- Profiling: perf on Linux, Instruments on macOS
- mmap versus read

**Wed, 2020-07-29 — Lachezar Lechev**

**“Drone Control - ‘Controlling a drone using Rust over WiFi’”**

# Recap: Drone Control Talk

https://github.com/AeroRust/Welcome

- TCP connections Handshake, establish connection
- JSON requests & responses
- ping-pong within 7s
- scroll crate: “A suite of powerful, extensible, generic, endian-aware Read/Write traits for byte buffers”

**Wed, 2020-07-29 — Nell Shamrell - Harrington**

**“The Rust Borrow Checker - A Deep Dive”**

# Recap: Drone Control Talk

Slides on slideshare

- “Is the Borrow Checker a friend or a foe?”
- Stages of Compilation:
  1. Lexical Analysis
  2. Parsing
  3. Semantic Analysis (Borrow Checker!)
  4. Optimization
  5. Code Generation
- BC tracks initializations/moves and applies lifetime inference
- Lifetime of a variable has two definitions
  - “Span of time before the value of a variable gets freed”
  - “scope of a variable”
- “If you make a reference to a value, the lifetime of that reference cannot outlive the scope of the value”
- `https://rustc-dev-guide.rust-lang.org/`

**Thu, 2020-07-30 - Jan-Erik Rediger**

**“Leveraging Rust to build cross-platform mobile libraries”**

# Recap: Mobile Libraries Talk

Slides on slideshare

- Firefox Telemetry project
- Collect performance metrics for our products, package pings at controlled schedules
- Three Principles
  - Stay Lean
  - Build Security
  - Engage Your Users
- Telemetry *scalars*: `Scalars.yaml` (metadata: bug_numbers, description, expires, notification_emails, …)

# Recap: Mobile Libraries Talk

- Glean core → Glean FFI → Glean Kotlin/Swift → Android/iOS app
- cbindgen crate: “A tool for generating C bindings to Rust code.”
- 10 Glean compilation targets supported
- Hello World with JNI (also see Otavio Pace’s talk)
- tagged unions are generated by bindgen for rust enums
- ProtoBuf to serialize data
- R8 minifies/optimizes JVM bytecode (successor to proguard)
- here: only invoke Rust code from Kotlin (never the other way around)

**Fri, 2020-07-31 - Luca Palmieri**

**“Are we observable yet? Telemetry for Rust APIs - metrics, logging, distributed tracing”**

# Recap: Telemetry for Rust APIs Talk

- Developer of DonateDirect
- New project: fast versus reliable (metrics, tracing, logs, …)
- Claim: convenience beats correctness
- Metrics give us an aggregate picture of the system state
- actix_web_prom crate provides a pluggable middleware with standard Prometheus metrics out of the box
- log crate and tracing crate to dump structured logging with JSON output and forward spans to elasticsearch and then kibana

# Recap: Telemetry for Rust APIs Talk

**Key takeaways:**

- Lack of telemetry is a ticking bomb
- Diagnostic instrumentation has to be easy
- Metrics to alert and monitor system state
- High cardinality is key to being able to detect and triage unknown unknowns
- Span as unit of work abstraction
- You must be able to trace a request across different services

**Fri, 2020-07-31 — Tim McNamara**

**“How 10 open source projects manage unsafe code”**

# Recap: Managing unsafe code Talk

**Unsafe guidelines for the impatient:**

- Use `#[deny(unsafe_code)]`
- Add comments to all unsafe blocks
- Let someone read the unsafe block comment.
  If they cannot explain afterwards, revise the comment

**Remarks:**

- Question `unsafe`! Look for safe alternatives
- rust std: “Unsafe code block need a comment explaining why they’re ok”
- “We try to create a situation where we, as a team, are building safe software and we are mentally switched on if we go to the unsafe module”
- UCG WG - Rust’s Unsafe Code Guidelines Working Group

# Recap: Managing unsafe code Talk

- `actix_web` incident
- `#[deny(unsafe_code)]` and `#[allow(unsafe_code)]` (can be nested)
- cargo-geiger "detects usage of unsafe Rust in a Rust crate and its deps", introduces `#![forbid(unsafe_code)]`
- exa uses syscalls natively, BLAKE3 uses SIMD instructions, Firecracker interacts with a hybervisor, librsvg talks to GLib, toolshed deals with pointers in one module, terminusdb interacts with Prolog, Fuchsia OS interacts with the non-rust kernel

# Dialogue

# Concurrency

# Definition: concurrency

*In computer science,* concurrency *is the ability of different parts or units of a program, algorithm, or problem to be executed out-of-order or in partial order, without affecting the final outcome.*

—*Concurrency (computer science)*

# Definition: parallelism

Parallel computing *is a type of computation in which many calculations or the execution of processes are carried out simultaneously.*

—*Parallel computing*

# Models of concurrency

Formally, different models exist:

- Parallel random-access machine
- Actor model
- Petri nets
- Process calculi (CCS, CSP, $\pi$-calculus)
- Tuple spaces
- Simple Concurrent Object-Oriented Programming (CSOOP)
- Reo Coordination Language

… to design distributed systems.

# Models of concurrency

For example, Erlang is famous for its distributed computational model.

But, in rust, we stick to a von-Neumann-like model:

- Instructions are executed in order.
- We have stacks, heaps, bss and data sections to organize the memory.
- We make syscalls to the kernel and compile against some ISA.

Thus it is a question of an API. We define *concurrent units* and they might run in *parallel*.

# Definition: concurrency

Concurrency as a question of level of granularity:

1. Instruction
2. blocks of instructions
3. Function
4. Several stacks, one heap
5. Process

In general: performance as incentive.

# Problems of concurrency

1. Data dependency (synchronization problem)

   **let** A and B be two concurrent units.
   Both want to increment $x$
   **A** reads that $x$ is 41
   **B** reads that $x$ is 41
   **A** increments $x$ and gets $y := 42$
   **B** increments $x$ and gets $z := 42$
   **A** writes $x := y$
   **B** writes $x := z$
   **x** is 42

2. Concurrent units need to exchange messages
3. Concurrent unit waits for an event
4. Interrupt concurrent unit (*preemption*)
5. Determine concurrent unit has finished

# In rust (from low-level to high-level)

# On instruction-level

# SIMD instructions

SIMD = **s**ingle **i**nstruction, **m**ultiple **d**ata

Run one instruction, apply arithmetic/logic/data-handling/memory instructions to several values simultaneously. 8/16/32/64/128/512

**RISC-V** I prefer to talk about the RISC-V ISA, but RISC-V basically dropped its Packed SIMD extension and develops a new "P" extension. Thus, I switch to x86_64.

**x86_64** Streaming SIMD Extensions (SSE) and Advanced Vector Extensions (AVX)

# SSE and AVX

Operands (put into XMM/YMM registers):

| | |
|---|---|
| **SSE** | four f32 |
| **SSE2** | two f64, two i64, four i32, eight i16, sixteen u8 |
| **SSE3** | (only 13 new instructions) |
| **SSE4** | (only 54 new instructions) |
| **AVX** | eight f32, four f64 |
| **AVX2** | 256-bit registers for almost everything |
| **AVX-512** | 512-bit registers but instructions split into multiple extensions |

# SIMD support

```
% cat /proc/cpuinfo | grep "sse\|avx"
flags   : fpu … sse sse2 … ssse3 … sse4_1 sse4_2
          … avx … avx2 … flush_l1d
flags   : fpu … sse sse2 … ssse3 … sse4_1 sse4_2
          … avx … avx2 … flush_l1d
flags   : fpu … sse sse2 … ssse3 … sse4_1 sse4_2
          … avx … avx2 … flush_l1d
flags   : fpu … sse sse2 … ssse3 … sse4_1 sse4_2
          … avx … avx2 … flush_l1d
```

# (No) SIMD instructions

```
fn foo(a: &[u8], b: &[u8], c: &mut [u8]) {
    for ((a, b), c) in a.iter().zip(b).zip(c) {
      println!("{} {} {}", a, b, c);
        *c = *a + *b;
    }
}

fn main() {
  let a: [u8; 4] = [0xDE, 0xAD, 0xBE, 0xEF];
  let b: [u8; 4] = [0x00, 0x01, 0x02, 0x03];
  let mut c = [0u8; 4];
  foo(&a, &b, &mut c);

  println!("{:?}", c);
}
```

# (No) SIMD instructions

On godbolt with `-C opt-level=1` (or 0):

```
.LBB58_2:
; …
mov     rdx, qword ptr [rsp + 24]
movzx   ecx, byte ptr [rcx]
add     cl, byte ptr [rax]
mov     byte ptr [rdx], cl
mov     rdi, r14
; …
jne     .LBB58_2
```

# SIMD instructions

On godbolt with `-C opt-level=2` (or higher):

```
movdqu  xmm2, xmmword ptr [rdx + rcx + 32]
paddb   xmm2, xmm0
movdqu  xmm0, xmmword ptr [rdx + rcx + 48]
paddb   xmm0, xmm1
movdqu  xmmword ptr [r8 + rcx + 32], xmm2
```

paddb: “add packed integer” instruction

# SIMD instructions

via std::arch ("SIMD and vendor intrinsics module"):

```
#[cfg(all(
  any(target_arch = "x86", target_arch = "x86_64"),
  target_feature = "avx2"
))]
fn foo() {
  #[cfg(target_arch = "x86")]
  use std::arch::x86::_mm256_add_epi64;
  #[cfg(target_arch = "x86_64")]
  use std::arch::x86_64::_mm256_add_epi64;

  unsafe {
    _mm256_add_epi64(...);
  }
}
```

# SIMD summary

- rust uses the LLVM stack, which implements *auto-vectorization*
- usually, we get SIMD instructions for free
- sometimes, you want to use the explicitly
- you can always use inline assembly with unsafe (`x86::time::rdtsc`)
- packed_simd provides a high-level API
- also: faster, ssimd
- mostly, libraries are specific for an application domain, e.g.:
  - numeric-array crate
  - directx_math crate
  - pqcrypto-classicmceliece crate

# On blocks of code

# On OS thread level

Copy stack (expensive?), share heap.

# Mutex

- “A mutual exclusion primitive useful for protecting shared data”
- A mutex can be `locked` (true/blocked) or released
- Also `try_lock` (true/false)
- Only one concurrent unit can hold a lock simultaneously
- If the mutex lock goes out of scope, another unit can acquire the lock.

# Mutex

```
use std::sync::Mutex;

fn main() {
    let val = Mutex::new(0u8);
    {
        *val.lock().unwrap() += 1;
        *val.lock().unwrap() += 1;
        *val.lock().unwrap() += 1;
    }
    println!("{}", val.lock().unwrap());
}
```

# Mutex

```rust
use std::sync::{Arc, Mutex};
use std::thread;

fn main() {
    let mutex = Arc::new(Mutex::new(0));
    let c_mutex = mutex.clone();

    thread::spawn(move || {
        *c_mutex.lock().unwrap() = 10;
    }).join().expect("thread::spawn failed");
    println!("{}", *mutex.lock().unwrap());
}
```

# async & await

# async & await

Similar to C# and JavaScript. `async` keyword.

```
async fn sub() -> u8 {
  42u8
}

fn main() {
  println!("{}", sub())
}
```

# async & await

```
meisterluk@gardner ~ % rustc --edition=2018 async1.rs
error[E0277]: `impl std::future::Future`
              doesn't implement `std::fmt::Display`
 --> async1.rs:6:20
  |
6 |     println!("{}", sub())
  |                    ^^^^^ `impl std::future::Future`
  |      cannot be formatted with the default formatter
  |
  = help: the trait `std::fmt::Display` is not implemented for
    `impl std::future::Future`
  = note: in format strings you may be able to use `{:?}`
    (or {:#?} for pretty-print) instead
  = note: required by `std::fmt::Display::fmt`
  = note: this error originates in a macro (in Nightly builds,
    run with -Z macro-backtrace for more info)

error: aborting due to previous error

For more information about this error, try `rustc --explain E0277`.
```

*A future represents a value, that is not ready yet. Eventually, the future resolves to a value.*
*—withoutboats in a Rust Latam talk 2019*

`await` keyword. Not `await X`, but `X.await`.

```
async fn sub() -> u8 {
    42u8
}

fn main() {
    println!("{}", sub().await)
}
```

# async & await

```
meisterluk@gardner ~ % rustc --edition=2018 async2.rs
error[E0728]: `await` is only allowed inside `async`
              functions and blocks
 --> async2.rs:6:20
  |
5 | fn main() {
  |    ---- this is not `async`
6 |     println!("{}", sub().await)
  |                    ^^^^^^^^^^ only allowed
  |         inside `async` functions and blocks

error: aborting due to previous error

For more information about this error, try `rustc --explain E0728`.
```

# async & await

So who can call the first async function? the *executor* (scheduler)

- e.g. async-std, smol, tokio
- executor is thus exchangable
- executor allocates memory per future
- futures are like state machines between states

the *reactor*:

- Executor: is future X ready to go? Yes, then go. Else:
- Reactor: I will take care of the future and report back when its ready

# async & await

```rust
use smol::io;

async fn sub() -> u8 {
  42u8
}


fn main() -> io::Result<()> {
    smol::run(async {
        println!("{}", sub().await);
        Ok(())
    })
    // prints 42
}
```

# async & await

```
async fn sub() -> u8 {
    42u8
}

fn main() {
    async {
        println!("{}", sub().await)
    };
}
```

# async & await

```
meisterluk@gardner ~ % rustc --edition=2018 async3.rs
warning: unused implementer of `std::future::Future`
         that must be used
 --> async3.rs:6:5
  |
6 | /     async {
7 | |         println!("{}", sub().await)
8 | |     };
  | |______^
  |
  = note: `#[warn(unused_must_use)]` on by default
  = note: futures do nothing unless you `.await`
          or poll them

warning: 1 warning emitted

meisterluk@gardner ~ % ./async3    # prints nothing
```

# async & await

Other async/await implementations:

- Calling an async function, schedules it
- Javascript calls *Promise*, what rust calls *Future*
- Green threads: a scheduler is compiled in every executable to manage small threads (M:N threading) (not OS threads!) (e.g. erlang, golang goroutines). rust 1.0 dropped green threads.

Rust implementations:

- Zero-cost: only await schedules the function (“lazy”)

futures, an official Rust crate that lives in the rust-lang repository A runtime of your choosing, such as Tokio, async$_s$*td*, *smol*, *etc*.

# async & await

```
fn get_two_sites() {
  // Spawn two threads to do work.
  let thread_one = thread::spawn(|| download("https://foo.com"));
  let thread_two = thread::spawn(|| download("https://bar.com"));

  // Wait for both threads to complete.
  thread_one.join().expect("thread one panicked");
  thread_two.join().expect("thread two panicked");
}
```

threading approach via async-book

# async & await

```
async fn get_two_sites_async() {
  // Create two different "futures" which, when run to completion,
  // will asynchronously download the webpages.
  let future_one = download_async("https://foo.com");
  let future_two = download_async("https://bar.com");

  // Run both futures to completion at the same time.
  join!(future_one, future_two);
}
```

async approach via async-book

# async & await summary

- Async-await is a way to write functions that can ”pause”, return control to the runtime, and then pick up from where they left off
- proposed 2016, didn’t make edition 2018, landed in rust 1.39
- Memory management
- wasm-bindgen-futures binds Rust Future to Javascript Promise

# On process-level

**But how many units?**

# How many concurrent units?

**How many?** Difficult.

- CPU cycles per instruction differs, caches everywhere!
- Thus, **benchmark, benchmark, benchmark!**

But how?

- No stable stdlib support in rust!
- RFC 29553: Tracking issue for [bench] and benchmarking support
- e.g. criterion crate

# How many concurrent units?

**Heuristics:**

- RFC 985: Add `std::env::concurrency_hint`
- `num_cpus` crate: `num_cpus::get();`
- `dupfiles-go` and ripgrep experience: the number of concurrent units reading files should roughly correspond to the number of logical CPUs

# Epilogue

# Quiz

**What does SIMD stand for?**

**What is a mutex?**

**What is a future?**

**Heuristically, what might be a good number of concurrent units?**

# Quiz

**What does SIMD stand for?**

Single Instruction, Multiple Data

**What is a mutex?**

**What is a future?**

**Heuristically, what might be a good number of concurrent units?**

# Quiz

**What does SIMD stand for?**

Single Instruction, Multiple Data

**What is a mutex?**

a data structure to give mutually exclusive access to a code section

**What is a future?**

**Heuristically, what might be a good number of concurrent units?**

# Quiz

**What does SIMD stand for?**

Single Instruction, Multiple Data

**What is a mutex?**

a data structure to give mutually exclusive access to a code section

**What is a future?**

represents a value, that is not ready yet

**Heuristically, what might be a good number of concurrent units?**

# Quiz

**What does SIMD stand for?**

Single Instruction, Multiple Data

**What is a mutex?**

a data structure to give mutually exclusive access to a code section

**What is a future?**

represents a value, that is not ready yet

**Heuristically, what might be a good number of concurrent units?**

number of logical CPUs of the machine

# Next time

| | |
|---|---|
| Next meetup | Wed, 2020/08/26 |
| Topic | I/O (files, file formats, simple TCP server) |

# Thank you!