Rust - Access All Areas

Florian Gilcher

goto Amsterdam 2019

CEO and Rust Trainer
Ferrous Systems GmbH

Joe Armstrong

- I read ”Programming Erlang” in 2009
- Learned from Joe’s how concepts, strictly applied, can help solve problems
- In this case: *shared nothing* and *message passing*
- Sometimes, you have to invent a language for it

Whoami

- Florian Gilcher
- https://twitter.com/argorak
- https://github.com/skade
- CEO https://asquera.de, https://ferrous-systems.com
- Rust Programmer and Trainer: https://rust-experts.com
- Mozillian
- Previously 10 years of Ruby community work

Whoami

- Started learning Rust in 2013
- Mostly out of personal curiosity
- Co-Founded the Berlin usergroup
- Organized RustFest and OxidizeConf
- Project member since 2015, mostly Community team, now Core, lead website team

Endorsements

I don't trust any programmer that deliberately uses Ruby or Java.

– Christopher Spencer, goto YouTube channel

I'm presenting the work of *180* team members and over *5000* contributors over the last years, culminating in a large release in December last year.

Rust was released in May 2015, and has been growing ever since.

A language empowering everyone to build reliable and efficient software.

The old trifecta

- Safe
- Concurrent
- Fast

The trifecta

- Performance
- Reliability
- Productivity

A mission

What do we need to bring that to everyone?

- A language that scales up and down
  - Small targets to large targets
  - High abstractions, low abstractions
- Useful Abstractions
  - That can be peeked through
  - Cost nothing
- Ergonomics and care
  - Unified tooling
  - Extensible tooling
  - Strict backwards compatibility

Programming problems to solve

- Memory safety
- Resource consumption
- Resource handling
- Concurrency and parallelism
- Dealing with external data
- Resilience
- Integration into existing code

Rust 2018 is a new *language profile* to enable all of the above.

- Enabled since December 2018
- No breaking change
- Old code is still fully supported

Area

Rust is

- A C/C++ Competitor...
- ... that is statically memory safe ...
- ... with features making it competitive with languages like Java, Scala and Go.

Model

Rust is

- A native programming language
- A *Values and Functions* language
- Ahead of time compiled
- Without active runtime
- Memory-Safe
- Generic
- Detailled error handling, no catchable exceptions

What if

What if we had a language that's a nitpicker, but in a good way?

Code example

```
struct Point {
    x: i32,
    y: i32,
}

fn main() {
    let point: Point = Point { x: 1, y: 1 };
    let heap: Box<Point> = Box::new(point);

    // look, no deallocation!
}
```

Ownership

- Any value introduced into a Rust program is *exlusively owned*
- Ownership can be *moved*
- When a value runs out of scope, *it is dropped*
- This moment is clearly defined

Code example

```
struct Point {
    x: i32,
    y: i32,
}

fn main() {
    let point: Point = Point { x: 1, y: 1 };
    let heap: Box<Point> = Box::new(point);

    // look, no deallocation!
}
```

Ownership Is Resource Management

- Rust values have a trackable region in memory where they are active
- This means, they can be used for resource management
- Ownership manages resources, *memory* is always one of them

Example: File management

```rust
use std::io::Read;
use std::fs::File;

pub fn read_file(path: &str) -> Result<String, std::io::Error> {
    let mut file: File = File::open(path)?;

    let mut buffer = String::new();
    file.read_to_string(&mut buffer)?;

    Ok(buffer)
}
```

When the file runs out of scope, it is also *closed automatically*. `?` is the *error handling operator*.

enums and Results

```
enum Result<T,E> {
    Ok(T),
    Err(E)
}

fn may_fail() -> Result<String, std::io::Error> {
    unimplemented!()
}

fn main() {
    match may_fail {
        Ok(string) => println!("worked: {}", string),
        Err(e) => println!("{:?}", e)
    }
}
```

Results are plain data.

Example: Reuse of a resource

```rust
use std::io::Read;
use std::fs::File;

pub fn print_file_and_close(mut file: File)
-> Result<(), std::io::Error> {
    let mut buffer = String::new();
    file.read_to_string(&mut buffer)?;

    println!("{}", buffer);

    Ok(())
}

fn main() -> Result<(), std::io::Error> {
    let file = File::open("Cargo.toml")?;
    print_file_and_close(file);
    print_file_and_close(file);
    Ok(())
}
```

Example: Reuse of a resource

```
error[E0382]: use of moved value: `file`
  --> examples/ownership_print.rs:16:26
   |
14 |     let file = File::open("Cargo.toml")?;
   |         ---- move occurs because `file` has type `std::fs::File`, which does not implement the `Copy` trait
15 |     print_file_and_close(file);
   |                          ---- value moved here
16 |     print_file_and_close(file);
   |                          ^^^^ value used here after move
```

Detour: Scope based management with closures

```ruby
def read_file()
    File::open("Cargo.toml") do |f|
        file.each_line { |l| puts l }
    end
end

read_file
```

Detour: Scope based management with closures

Let's break things!

```
def read_file()
    iter = nil;

    File::open("Cargo.toml") do |f|
        iter = f.each_line
    end

    iter.each { |l| puts l }
end

read_file
```

Oops.

```
examples/read_file_broken.rb:8:in `each_line': closed stream (IOError)
        from examples/read_file_broken.rb:8:in `each'
        from examples/read_file_broken.rb:8:in `read_file'
        from examples/read_file_broken.rb:11:in `<main>'
```

What happened

- We’re referencing the file through an iterator
- We move the iterator out of the scope
- The file is closed
- We try to iterate -> BANG!

Let’s try this in Rust!

Rust also has references!

```
use std::io::BufReader;
use std::fs::File;

pub fn read_file(path: &str) -> Result<BufReader<&mut File>, std::io::Error> {
    let mut file: File = File::open(path)?;

    let reader: BufReader<&mut File> = BufReader::new(&mut file);

    Ok(reader)
}

fn main() {
    read_file("examples/ownership_file.rs");
}
```

Let’s try this in Rust!

```
error[E0515]: cannot return value referencing local variable `file`
 --> counter_examples/ownership_breakage.rs:9:5
  |
7 |     let reader: BufReader<&mut File> = BufReader::new(&mut file);
  |                                                       --------- `file` is borrowed here
8 |
9 |     Ok(reader)
  |     ^^^^^^^^^^ returns a value referencing data owned by the current function

error: aborting due to previous error

For more information about this error, try `rustc --explain E0515`.
```

Conclusion

- We’re referencing the file through a buffered reader
- We move the reader out of the scope by returning
- The file is closed, because the scope ends
- The compiler detects this as illegal

Borrowing

References in Rust are subject to a system called Borrowing

- References cannot *outlive* what they are borrowed from
- Mutable and immutable references cannot alias
- Mutable references have to be *unique*
- References are always valid

Mutable state and *shared state* in Rust is allowed, but not *shared mutable state*.

Illegal states

Files in Rust are defined in such a way that they are always open.

Rust gives you methods to *make illegal state irrepresentable*. Even if you wanted a File API that represents both open and closed, it would allow you to define which API is legal in both cases.

In contrast, Ruby's and other languages approach is about not forgetting the *close* call.

Borrowing

```rust
use std::io::prelude::*;
use std::io::BufReader;
use std::fs::File;

pub fn read_file(path: &str) -> Result<BufReader<File>, std::io::Error> {
    let file: File = File::open(path)?;

    let reader: BufReader<File> = BufReader::new(file);

    Ok(reader)
}

fn main() -> Result<(), std::io::Error> {
    let source = read_file("Cargo.toml")?;

    let mut i = 0;
    for line in source.lines() {
        i += 1;
        println!("{}: {}", i, line?);
    }
```

Important

- Ownership here is strict: BufReader now *owns* the file
- No one else has access to the file during that time!

Another tri

Rust APIs often come in threes:

- Owned
- Borrowed
- Mutably borrowed

Example: Iterators

```
fn main() {
    let vec = vec![1,2,3];
    let iter = vec.into_iter();

    for i in iter {
        println!("{}", i);
    }
}
```

Example: Iterators

```
fn main() {
    let vec = vec![1,2,3];
    let iter = vec.iter();

    for i in iter {
        println!("{}", i);
    }
}
```

Example: Iterators

```
fn main() {
    let mut vec = vec![1,2,3];
    let iter = vec.iter_mut();

    for i in iter {
        *i += 1;
    }

    println!("{:?}", vec) // [2, 3, 4]
}
```

Parallel Processing

- Great frameworks for parallel programming
- Mixable with concurrent approaches
- Safe from data races

Parallel Processing: Example

```
fn sum_of_squares(input: &[i32]) -> i32 {
    input.iter() // <-- just change that!
         .map(|&i| i * i)
         .sum()
}

fn main() {
    sum_of_squares(&[1,2,3]);
}
```

Parallel Processing: Example

```
use rayon::prelude::*;

fn sum_of_squares(input: &[i32]) -> i32 {
    input.par_iter()
         .map(|&i| i * i)
         .sum()
}

fn main() {
    sum_of_squares(&[1,2,3]);
}
```

Boring, isn't it?

Parallel Processing: Libraries

- crossbeam, base types for async: https://github.com/crossbeam-rs/crossbeam
- rayon, easy parallel processing: https://github.com/rayon-rs/rayon

Concurrent Programming

- Multiple frameworks for concurrent programming
- Mixable with parallel approaches
- Safe from data races

Concurrent Programming: Currently

```
fn main() {
    let addr = "127.0.0.1:7878".parse().unwrap();
    let listener = TcpListener::bind(&addr).unwrap();

    let server = listener.incoming().map_err(|err| {
            println!("stream error = {:?}", err);
        }).for_each(|socket| {
            let buffer = Vec::new();

            read_to_end(socket, buffer).and_then(|(socket, buffer)| {
                let s = String::from_utf8(buffer).unwrap();
                let parsed = protocol::parse(&s).unwrap();
                println!("{:?}", parsed);
                Ok(())
            })

        .map_err(|err| {
            println!("reading error = {:?}", err);
        })
    })
```

This is workable, but verbose and very error-prone.

```rust
#[runtime::main]
async fn main() -> Result<(), ServerError> {
    let mut incoming = {
        // set up a TCP server...
    };
    let rced_storage = Arc::new(Mutex::new(Vec::new()));

    while let Some(stream) = incoming.next().await {
        let storage = rced_storage.clone();

        runtime::spawn(async move {
            handle(stream?, &storage).await?;

            Ok::<(), ServerError>(())
        }).await?;
    }

    Ok(())
}
```

- actix and actix web: https://actix.rs/
- tokio/romio: concurrent event reactors
- runtime library facade: https://github.com/rustasync/runtime

Aside: Rust Send & Sync

Rust controls concurrency through 2 additional properties: *Send* & *Sync*.

- *Send* means that data can be *passed* between concurrent units
- *Sync* means that data can be *shared* between concurrent units

Both properties are independent of the parallelism or concurrency library in use.

Example: Threading

```
struct Counter {
    count: u32
}

fn main() {
    let mut counter = Counter { count: 0 };

    for _ in 1..=3 {
        std::thread::spawn(move || {
            counter.count += 1
        });
    }
}
```

Example: Threading

```
error[E0382]: use of moved value: `counter`
  --> examples/threading_error.rs:9:28
   |
6  |     let mut counter = Counter { count: 0 };
   |         ----------- move occurs because `counter` has type `Counter`, which does not implement the `Copy`
trait
...
9  |         std::thread::spawn(move || {
   |                            ------- value moved into closure here, in previous iteration of loop
10 |             counter.count += 1
   |             ------- use occurs due to use in closure
```

Example: Threading

```
use std::rc::Rc;
struct Counter {
    count: u32
}

fn main() {
    let mut counter = Rc::new(Counter { count: 0 });

    for _ in 1..=3 {
        let thread_handle = counter.clone();
        std::thread::spawn(move || {
            thread_handle.count += 1
        });
    }
}
```

Example: Threading

```
error[E0277]: `std::rc::Rc<Counter>` cannot be sent between threads safely
  --> examples/threading_error_rc.rs:11:9
   |
11 |         std::thread::spawn(move || {
   |         ------------------ `std::rc::Rc<Counter>` cannot be sent between threads safely
   |
```

Example: Threading

```
use std::sync::Arc;
struct Counter {
    count: u32
}

fn main() {
    let mut counter = Arc::new(Counter { count: 0 });

    for _ in 1..=3 {
        let mut thread_handle = counter.clone();
        std::thread::spawn(move || {
            thread_handle.count += 1
        });
    }
}
```

Example: Threading

```
error[E0594]: cannot assign to data in a `&` reference
  --> examples/threading_error_arc.rs:12:13
   |
12 |             thread_handle.count += 1
   |             ------------------------ cannot assign
```

Example: Threading

```
use std::sync::{Arc,Mutex, MutexGuard};
struct Counter {
    count: u32
}

fn main() {
    let counter = Arc::new(Mutex::new(Counter { count: 0 }));

    for _ in 1..=3 {
        let thread_handle = counter.clone();
        std::thread::spawn(move || {
            let mut lock: MutexGuard<_> =
                thread_handle.lock().unwrap();
            lock.count += 1
        });
    }
}
```

Practical examples

CLI programs

- Shippable without runtime
- Memory-conserving with fast startup time
- Fast and convenient parsers
- Free choice of concurrency patterns
- Ownership makes external resource management easy

CLI: Code Example

```rust
use structopt::StructOpt;

#[derive(StructOpt)]
struct Cli {
    /// The pattern to look for
    pattern: String,
    /// The path to the file to read
    #[structopt(parse(from_os_str))]
    path: std::path::PathBuf,
}

fn main() -> Result<(), std::io::Error> {
    let args = Cli::from_args();
    let content = std::fs::read_to_string(&args.path)?;

    for line in content.lines() {
        if line.contains(&args.pattern) {
            println!("{}", line);
        }
    }
```

CLI: Serialization/Deserialization

```rust
use serde::{Serialize, Deserialize};
#[derive(Serialize, Deserialize, Debug)]
struct Point { x: i32, y: i32 }

fn main() {
    let point = Point { x: 1, y: 2 };

    let serialized = serde_json::to_string(&point).unwrap();
    // Prints serialized = {"x":1,"y":2}
    println!("serialized = {}", serialized);

    let deserialized: Result<Point, _> = serde_json::from_str(&serialized);
    // Prints deserialized = Ok(Point { x: 1, y: 2 })
    println!("deserialized = {:?}", deserialized);
}
```

Type-informed serialization and deserialization, generated at compile-time!

```rust
#[derive(Serialize, Deserialize, Debug)]
#[serde(rename(serialize = "point"))]
struct Point {
    #[serde(default)]
    x: i32,
    #[serde(default)]
    y: i32,
}
```

Opt-In customization, including custom deserialization code.

- No runtime overhead
- Great cross-compiling support

Embedded Linux: Fields of use

- IoT Gateways
- Home routers
- Industry control systems
- Cars?

```
$ rustup target install aarch64-unknown-linux-musl
$ cargo build --target aarch64-unknown-linux-musl
```

As long as a target platform linker and libc is available.

Bare-Metal embedded

- Rust stabilized bare metal embedded support in 2018
  - Stabilisation of all low-level details: replacable error handlers etc.
- Great support for safe patterns on embedded devices
- Relies on existing tooling
- Should be considered young, but solid

Example: Minimal Embedded Rust

```
//! Minimal `cortex-m-rt` based program

#![deny(unsafe_code)]
#![deny(warnings)]
#![no_main]
#![no_std]

extern crate cortex_m_rt as rt;
extern crate panic_halt;

use rt::entry;

// the program entry point
#[entry]
fn main() -> ! {
    loop {}
}
```

Example: Memory mapping

```
mod syst {
    #[repr(C)]
    pub struct RegisterBlock {
        /// Control and Status
        pub csr: RW<u32>,
        /// Reload Value
        pub rvr: RW<u32>,
        /// Current Value
        pub cvr: RW<u32>,
        /// Calibration Value
        pub calib: RO<u32>,
    }
}

fn place_syst() -> *const syst::RegisterBlock {
    0xE000_E010 as *const _
}
```

Using Rust semantics

- Rust on Embedded uses *Ownership* to handle device access
- Uses *metaprogramming* facilities to provide convenience
- Resulting code is board-specific

Bare-Metal embedded: Options

- Directly use a board package
  - see https://github.com/rust-embedded for supported boards
- Use RTFM: https://github.com/japaric/cortex-m-rtfm
- Use a full embedded operating system: https://www.tockos.org/
- It's possible to use Rust on top of C-based embedded OSes like RIOT

Missing

Flashing and debugging with Rust or fully Rust-integrated targets. This is not a language problem!

Shared library use

- Rust can generate static an dynamic libraries
- Punctual speedup of larger programs
- Code sharing between different platforms
- Classic C usecase, reuses infrastructure
- Often used on mobile for shared libraries between Android an iOS

Shared library use: Example

```
#[derive(Debug)]
#[repr(C)]
pub struct Point {
    x: i32,
    y: i32
}

#[no_mangle]
pub extern "C" fn new_point(x: i32, y: i32) -> *mut Point {
    let p = Box::new(Point { x: x, y: y });
    Box::into_raw(p)
}

#[no_mangle]
pub extern "C" fn destroy_point(p: *mut Point) {
    unsafe { Box::from_raw(p) };
}

#[no_mangle]
pub extern "C" fn inspect_point(p: &mut Point) {
```

Shared library use: Example

```ruby
require 'ffi'

class Point < FFI::Struct
  layout  :int32, :int32
end

module LibPoint
  extend FFI::Library
  ffi_lib './libpoint.so'
  attach_function :new_point, [ :int32, :int32 ], :pointer
  attach_function :destroy_point, [ :pointer ], :void
  attach_function :inspect_point, [ :pointer ], :void
end

ptr = LibPoint.new_point(1, 1)
point = Point.new ptr
LibPoint.inspect_point point.pointer
LibPoint.destroy_point point.pointer
```

Shared library us

Generators

- Rust program -> C/C++ Header
- C/C++ program -> Rust bindings
- Specialized tooks for Python/Ruby/Node
- Rust program -> WASM

WebAssembly - WASM

- A machine independent binary format that can be run in a sandbox
- Almost as efficient as native code
- Rust is the prime langage for it

WebAssembly - WASM: Example

```rust
#[wasm_bindgen]
extern {
    fn alert(s: &str);
}

#[wasm_bindgen]
pub fn greet() {
    alert("Hello, wasm-game-of-life!");
}
```

WebAssembly - WASM: Example

- This is compiled through the standard cross-compilation toolchain
- Additional post-processing to generate a JS layer for direct access

WebAssembly - WASM: Example (JS Side)

```
import * as wasm from './wasm_hello_world';

export function greet() {
    return wasm.greet();
}
```

- WebAssembly is currently a minimum viable product
- A lot of things are down the road

WebAssembly - WASM: Users

- Cloudflare Workers: https://workers.cloudflare.com/
- Deployed in all major browsers

There's an extensive community around network programming.

- A zero-allocation userlevel TCP stack: https://github.com/m-labs/smoltcp
- Sōzu, a Rust reverse proxy: https://github.com/sozu-proxy/sozu
- LinkerD, a service mesh for microservices: https://linkerd.io/

Other usecases

- Gaming
  - Embark Studios, Amethyst Game engine
- Infrastructure Software
  - Amazon Firecracker
  - Chef Habitat

Missing: Safety Critical

- Formal proofing of the base language
- Certification of the compiler next?
- Sealed Rust: an attempt to bring certification of the Rust compiler on track

https://ferrous-systems.com/blog/sealed-rust-the-pitch/

The trifecta

- Performance
- Reliability
- Productivity

Performance

- A language as fast as C/C++
- With safety while doing the fast thing
- Abstractions with no overhead

Reliability

- Rust allows expression of complex abstract concepts...
- ... on a close-to-the metal basis ...
- ... with type-level support for resource management.

Productivity

- Great tooling
- Great documentation: https://rust-lang.org/learning
  - extensive stdlib docs
  - 9 books: language, embedded, cli tooling, internals...
- A language well feasible for performance refactoring

Programming problems to solve

- Memory safety - through the type system
- Resource consumption - by working with values and references
- Resource handling - through Ownership
- Concurrency and parallelism - through the type system
- Dealing with external data - through type informed frameworks
- Resilience - making illegal state irrepresentable
- Integration into existing code - through C integration

> *General reminder that if you encounter a* rustc *diagnostic error that confuses you for more than a minute, it is a* bug. *File tickets, we take them seriously. We want* rustc *to be your first tutor.*

– Estaban Küber, responsible for diagnostics

Does that work?

- We're seeing Rust used in many ways
- Roughly 33% influx from each
  - Functional languages
  - Dynamic languages
  - Systems languages

The cost of switching languages

We always consider *switching language* a high cost, while bringing *another tool to the belt* is cheap.

Conclusion

Rust is a language with well-chosen compile-time guarantees and simple runtime semantics that allows you to use it in *any* area of your product.

Thank you!

- https://twitter.com/argorak
- https://github.com/skade
- https://speakerdeck.com/skade
- florian.gilcher@ferrous-systems.com
- https://ferrous-systems.com
- https://rust-experts.com