Considering Rust

February 2020

Jon Gjengset

Graduate student at MIT's Parallel and Distributed Operating Systems group.

Why me?

- **Noria:** 70k LOC Rust database
- Contributor to Rust std and async runtime
- 150+ hours of Rust live-coding streams
- Experience in C, C++, Go, and Python, as well as familiarity with Java

Getting in touch:

- jon@tsp.io
- https://tsp.io
- https://twitter.com/jonhoo
- https://github.com/jonhoo
- https://youtube.com/c/JonGjengset

Why we’re all here today

Things we **will** cover:

- High-level comparisons with Java, C++, and Python
- Rust’s major selling points
- Rust’s primary drawbacks
- Long-term viability

Things we **will not** cover:

- Learning to program in Rust

First, meet Rust

The buzzwords

- By Mozilla, for “systems programming”
- “Fast, reliable, productive — pick three”
- “Fearless concurrency”
- Community-driven and open-source

The technical bits

- Compiled language (not bytecode — machine code)
- Strong, static typing
- Imperative, but with functional aspects
- No garbage collection or runtime
- Elaborate type system

Quick comparison

vs Python

Much faster.

Much lower memory use.

Multi-threading.

Algebraic data types.

Pattern matching.

Comprehensive static typing, so:

Many fewer runtime crashes.

vs Java

No JVM overhead or GC pauses.

Much lower memory use.

Zero-cost abstractions.

ConcurrentModificationException

Pattern matching.

Unified build system.

Dependency management.

vs C/C++

No segfaults.

No buffer overflows.

No null pointers.

No data races.

Powerful type system.

Unified build system.

Dependency management.

vs Go

No GC pauses; lower memory use.

No null pointers.

Nicer error handling.

Safe concurrency.

Stronger type system.

Zero-cost abstractions.

Dependency management.

Rust’s primary features

Modern language

or: it’s nice to use

Nice and efficient generics.

Algebraic data types + patterns.

Modern tooling.

Modern language

or: it’s nice to use

Nice and efficient generics.

Algebraic data types + patterns.

Modern tooling.

```rust
struct MyVec<T> {
  // ...
}

impl<T> MyVec<T> {
  pub fn find<P>(&self, predicate: P) -> Option<&T>
   where P: Fn(&T) -> bool
  {
    for v in self {
      if predicate(v) {
        return Some(v);
      }
    }
    None
  }
}
```

Nice and efficient generics.

Modern language

Nice and efficient generics.

Algebraic data types + patterns.

Modern tooling.

```
// Option<T> is an enum that is either Some(T) or None
if let Some(f) = my_vec.find(|t| t >= 42) {
  /* found */
}
```

Algebraic data type + patterns

```
// Option<T> is an enum that is either Some(T) or None
if let Some(f) = my_vec.find(|t| t >= 42) {
  /* found */
}

enum DecompressionResult {
  Finished { size: u32 },
  InputError(std::io::Error),
  OutputError(std::io::Error),
}

// this will not compile:
match decompress() {
  Finished { size } => { /* parsed successfully */ }
  InputError(e) if e.is_eof() => { /* got EOF */ }
  OutputError(e) => { /* output failed with error e */ }
}
```

Algebraic data type + patterns

Modern language

Nice and efficient generics.

Algebraic data types + patterns.

Modern tooling.

```rust
#[test]
fn it_works() {
  assert_eq!(1 + 1, 2);
}

/// Returns one more than its argument.
///
/// ```
/// assert_eq!(one_more(42), 43);
/// ```
pub fn one_more(n: i32) -> i32 {
  n + 1
}
```

Modern tooling — built-in testing and docs; friendly errors

Modern language

Nice and efficient generics.

Algebraic data types + patterns.

Modern tooling.

Safety by construction

or: it’s harder to misuse

Pointers checked at compile-time.

Thread-safety from types.

No hidden states.

Safety by construction

or: it’s harder to misuse

Pointers checked at compile-time.

Thread-safety from types.

No hidden states.

```
// every value has an owner, responsible for destructor (RAII).
// compiler checks:

// only ever one owner:
// no double-free

let x = Vec::new();
let y = x;
drop(x); // illegal, y is now owner

// no pointers live past owner changes or drops:
// no dangling pointers/use-after-free

let mut x = vec![1, 2, 3];
let first = &x[0];
let y = x;
println!("{}", *first); // illegal, first became invalid when x was moved
```

Pointers checked at compile-time.

```
let v = Vec::new();

// this compiles just fine:
println!(“len: {}”, v.len());

// this will not compile; would need mutable access
v.push(42);
```

Pointers checked at compile-time.

```rust
let v = Vec::new();
accidentally_modify(&v);

fn accidentally_modify(v: &Vec<i32>) {
  // this compiles just fine:
  println!("len: {}", v.len());
  // this will not compile; would need &mut Vec<i32>
  push(v);
}

// explicitly declare need for mutable access
fn push(v: &mut Vec<i32>) {
  v.push(42);
}

// this will not compile either:
push(&mut v);
```

Pointers checked at compile-time.

Safety by construction

Pointers checked at compile-time.

Thread-safety from types.

No hidden states.

```rust
use std::cell::Rc; // reference-counted, non atomic
use std::sync::Arc; // reference-counted, atomic

// this will not compile:
let rc = Rc::new("not thread safe");
std::thread::spawn(move || { println!("I have an rc with: {}", rc); });

// this compiles fine:
let arc = Arc::new("thread safe");
std::thread::spawn(move || { println!("I have an arc with: {}", arc); });

// this will also not compile:
let mut v = Vec::new();
std::thread::spawn(|| { v.push(42); });
let _ = v.pop();
```

Thread-safety embedded in type system.

Safety by construction

Pointers checked at compile-time.

Thread-safety from types.

No hidden states.

```
enum Option<T> {
  Some(T),
  None,
}
enum Result<T, E> {
  Ok(T),
  Err(E),
}

// v is Option<&T>, not &T -- cannot use without checking for None
let v = my_vec.find(|t| t >= 42);

// n is Result<i32, ParseIntError> -- cannot use without checking for Err
let n = “42”.parse();

// ? suffix is “return Err if Err, otherwise unwrap Ok”
let n = “42”.parse()?;
```

No hidden states.

Safety by construction

Pointers checked at compile-time.

Thread-safety from types.

No hidden states.

Low-level control

or: it gets out of your way

No GC or runtime.

Control allocation and dispatch.

Can write + wrap low-level code.

Low-level control

or: it gets out of your way

No GC or runtime.

Control allocation and dispatch.

Can write + wrap low-level code.

That means:

No garbage collection pauses.

No memory overhead (except what you add).

Can issue system calls (incl. fork/exec).

Can run on systems without an OS.

Free FFI calls to other languages.

No GC or runtime.

Low-level control

No GC or runtime.

Control allocation and dispatch.

Can write + wrap low-level code.

```rust
// variables are all on the stack
let x = 42;
let z = [0; 1024];

// can opt-in to heap allocation
let heap_x = Box::new(x);
let heap_z = vec![0; 1024];

// can swap out allocator for performance or on embedded
#[global_allocator]
static A: MyAllocator = MyAllocator;

// can opt-in to dynamic dispatch (vtable): only one copy of find per T
impl<T> MyVec<T> {
  pub fn find(&self, f: &dyn Fn(&T) -> bool) -> Option<&T> {
    // ...
```

Control allocation and dispatch.

Low-level control

No GC or runtime.

Control allocation and dispatch.

Can write/wrap low-level code.

```
// can do highly unsafe things by marking them as such
let vga_ptr = 0xB8000 as *mut u8;

// assuming we have exclusive access to VGA memory
let vga = unsafe { std::slice::from_raw_parts_mut(vga_ptr, 80 * 24) };
vga[0] = b'X';

// back in safe code, usual rules apply:
*vga_ptr = b'Y'; // invalid; trying to dereference raw pointer
vga[80 * 24] = b'Y'; // bounds check will catch this
```

Can write and wrap low-level code.

Low-level control

No GC or runtime.

Control allocation and dispatch.

Can write + wrap low-level code.

Compatibility

or: it plays nicely with others

Zero-overhead FFI.

Great WebAssembly support.

Works with traditional tools.

Compatibility

or: it plays nicely with others

Zero-overhead FFI.

Great WebAssembly support.

Works with traditional tools.

```rust
// trivial to get access to any function following C ABI
extern "C" {
  fn c_abs(input: i32) -> i32;
}

fn main() {
  // inherently unsafe; who knows what C code does
  println!(“{}”, unsafe { c_abs(-42) });
}

// trivial to expose Rust functions/types through C ABI
#[no_mangle]
pub extern "C" fn callable_from_c() {
    println!("This function is callable from C");
}
```

Zero-overhead FFI.

Compatibility

Zero-overhead FFI.

Great WebAssembly support.

Works with traditional tools.

Compatibility

Zero-overhead FFI.

Great WebAssembly support.

Works with traditional tools.

Rust uses LLVM, normal calling conventions, no runtime, DWARF, so:

perf works.

gdb/lldb works.

valgrind works.

LLVM sanitizers work.

Works with traditional tools.

Compatibility

Zero-overhead FFI.

Great WebAssembly support.

Works with traditional tools.

Tooling

or: it comes with batteries

Dependency management.

Standard tools included.

Excellent support for macros.

Tooling

or: it comes with batteries

Dependency management.

Standard tools included.

Excellent support for macros.

```
# Cargo.toml
[dependencies]
regex = “1.3.3”
rayon = “1.2”
csv = { git = “https://github.com/…” }

[dev-dependencies]
quickcheck = “0.9.2”

$ cargo build
Downloading regex 1.3.4
Downloading rayon 1.3.0
Updating git repository ‘https://github.com/…’
Compiling ...
$ cargo test
Downloading quickcheck 0.9.2
Compiling ...
Running ...
```

Built-in dependency management.

Tooling

Dependency management.

Standard tools included.

Excellent support for macros.

Standard distribution ships with:

cargo fmt — code formatter

cargo doc — documentation generator

cargo clippy — linter

rls/rust-analyzer — compiler front-end for IDE integration

Standard tools included.

Tooling

Dependency management.

Standard tools included.

Excellent support for macros.

```
macro_rules! assert_eq {
  ($left:expr, $right:expr) => {
    let left = $left;
    let right = $right;
    if left != right {
      panic!("assertion failed: {:?} != {:?}", left, right);
    }
  }
}

assert_eq!(1 + 1, 2);
```

Excellent support for macros.

```
macro_rules! assert_eq {
  ($left:expr, $right:expr) => {
    let left = $left;
    let right = $right;
    if left != right {
      panic!("assertion failed: {:?} != {:?}", left, right);
    }
  }
}

assert_eq!(1 + 1, 2);


#[derive(Serialize, Deserialize)]
struct Person {
  name: String,
  age: u32,
}
```

Excellent support for macros.

Tooling

Dependency management.

Standard tools included.

Excellent support for macros.

Asynchronous code

Language support for writing asynchronous code.

Choose your own runtime!

Watch this space — still evolving.

Rust’s primary drawbacks

Learning curve

The borrow checker is different.

No object-oriented programming.

Ecosystem

Young, and few maintainers.

Small (but growing quickly).

No runtime

No runtime reflection.

No runtime-aided debugging.

Compile time

Improving, but still slow.

No pre-built libraries.

Vendor support

Huge C++ libraries are a pain.

Tooling that only supports C++.

Windows

Full compiler/std support.

Limited library support.

Long-term viability

Long-term viability seems high.

- Most loved language four years running.
- Adoption by large companies (“Friends of Rust”):
  - Mozilla, Dropbox, CloudFlare, Microsoft, Google, Amazon, Facebook, Atlassian, npm
- Great interoperability story; easy incremental adoption.
- Increased company involvement in Rust itself.
- ~10 yearly conferences around the world.

For more:

- https://matklad.github.io/2020/02/14/why-rust-is-loved.html
- https://twitter.com/jonhoo/status/1205184012861497344
- https://twitter.com/jonhoo/status/1215739214576287744

Slide license