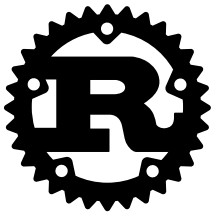


Rust programming language

a systems language

pursuing the trifecta

safe, concurrent (?), fast

Introduction to safe systems programming

Quick introduction

- Mozilla Research project
- Started by Graydon Hoare in 2006
- 1.0-alpha just released, 1.0 set to be released in **early March**
- Main niches:
  - Large scale maintainable systems: browser engines, games, databases
  - Fast, *predictably fast* code
  - Correctness and confidence in the quality
- Substantial influence of functional languages
- Still, striving to be *close to the machine*

```rust
fn main() {
    println!("Hello world!");
}
```

Run

Run

```
fn main() {
    // A simple integer calculator:
    // + or - means add or subtract by 1
    // * or / means multiply or divide by 2

    let program = "+ + * - /";
    let mut accumulator = 0i;

    for token in program.chars() {
        match token {
            '+' => accumulator += 1,
            '-' => accumulator -= 1,
            '*' => accumulator *= 2,
            '/' => accumulator /= 2,
            _ => { /* ignore everything else */ }
        }
    }

    println!("The program \"{}\" calculates the value {}",
             program, accumulator);
}
```

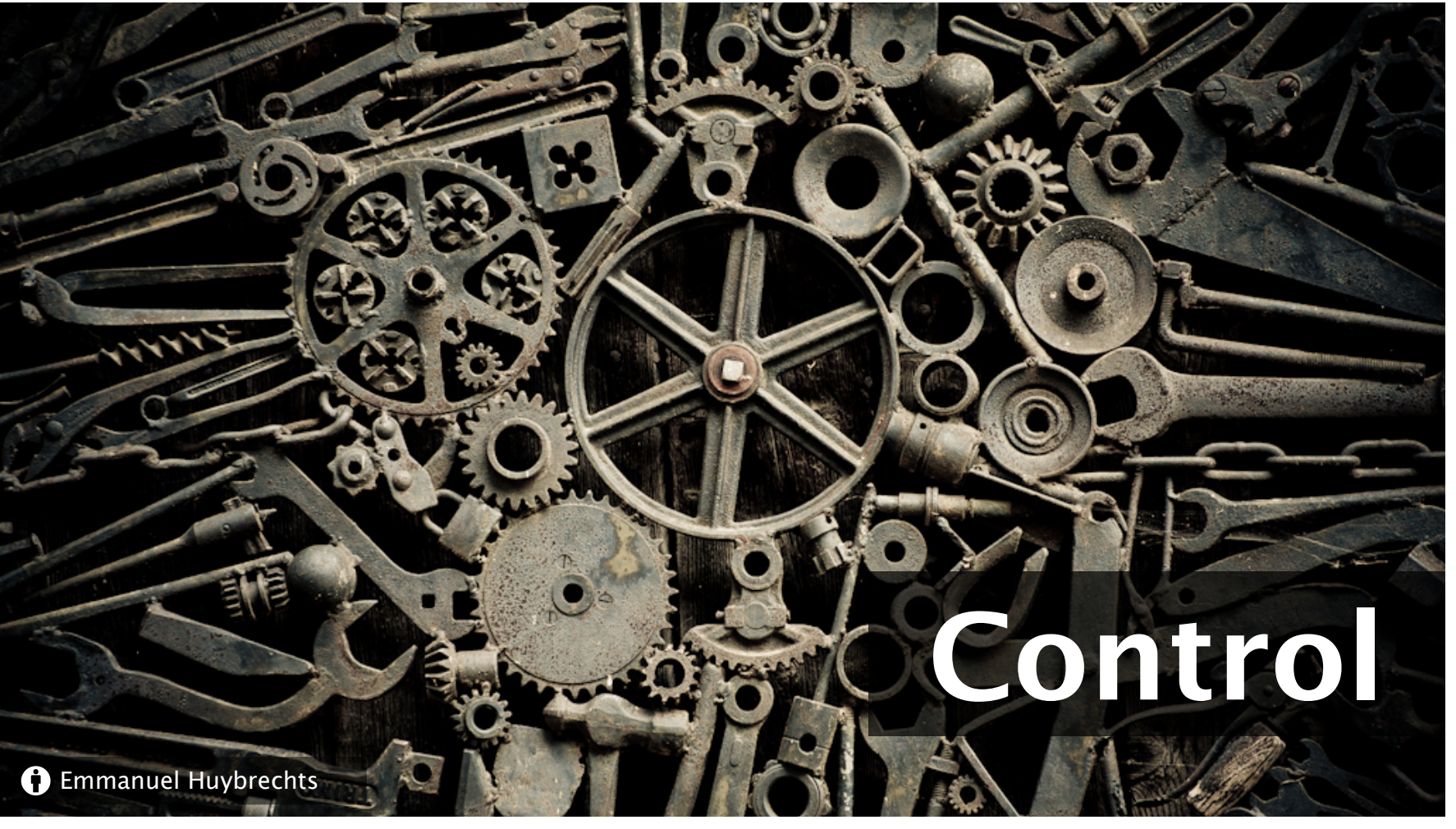
What choice do programmers have today?

Close to the metal but easy to get wrong

- C
- C++

High level, secure but with less control

- Java
- Scala
- Python
- Ruby
- JavaScript
- Haskell


Control
Emmanuel Huybrechts

- Deterministic resource management: *memory*, *sockets*, *locks*
  - Resource Acquisition Is Initialisation

```
use std::io::File;
fn main() {
    let file = File::open(&Path::new("/Users/jakub/TODO.txt"));
    // file closed when it goes out of scope.
}
```

Run

- Machine-sympathetic memory layout

```
use std::intrinsics::transmute;
struct Point { x: u32, y: u32 }
struct Rectangle {
    top_left: Point,
    width: u32,
    height: u32
}
fn main() {
    let rectangle = Rectangle {
        top_left: Point { x: 0, y: 0 },
        width: 50,
        height: 50
    };

    println!("{}\n{}",
        &rectangle.top_left.y as *const u32 as u64,
        &rectangle.width as *const u32 as u64);
}
```

Run

- Zero-overhead memory abstractions

```
struct Point {
    x: u32,
    y: u32
}
struct Rectangle {
    top_left: Point,
    width: u32,
    height: u32
}
fn main() {
    println!("{}", std::mem::size_of::<Rectangle>());
}
```

Run

- Can use on bare metal (no runtime).
- Can interoperate with C libraries.
- Statically determined memory deallocation.

```
fn print_and_forget(x: String) {
    println!("{}", x);
    // free(x);
}
```

Run

Safety

daspunkt

Classes of errors common in C++

- Dangling pointers
- Out-of-bounds array accesses
- Memory leaks
- Buffer overrun
- Use-after-free
- Data races
- Iterator invalidation

Classes of errors common in Java

- NullPointerException
- ConcurrentModificationException
- Out of heap


kasp
12550
Security
Rob

Security hazards

Security Advisories For Firefox

We don't want any of this.

- *Best practices* are not enough.
- Enforcing safety via idioms is not working.
- The compiler should reject programs that are not safe.


Speed
Jason Devaun

Performance

- Very good performance - on-par with C and C++
  - State-of-the-art optimization pipeline using LLVM
- *Predictable* performance
  - Ahead-of-time compilation to a static binary
  - No garbage collection
  - Static dispatch whenever possible
- Very early days
  - LLVM driven with C, C++ frontends in mind
  - The Rust compiler knows more about your programs
- Computer Language Benchmarks Game


Can we have the cake and eat it too?
Jason Devaun

Yes.

- Rust guarantees memory safety with strong compile-time analysis.
- Strong static type system
- No explicit memory management, the compiler knows *precisely* when memory can and must be deallocated.
- If a Rust program compiles, it's memory safe.
- Zero-cost abstraction characteristics of a C++ program.
- Type inference makes programs easy to write and read.

Main idea

Ownership

- Who is allowed to release a resource? *Memory*, in particular.
- Which thread can safely access a resource?
- When can a resource be safely *moved* in memory?

```
fn do_things(x: Box<uint>) {
    println!("{0} + {0} = {1}", x, *x + *x);
}

fn main() {
    let foo = box 42u;
    do_things(foo);
    println!("{}", foo);
}
```

Run

- A datum is owned by one owner at a time.

Borrowing

- Who has read access to a resource but doesn't own it?
- Who has mutable access to a resource?
- Cannot lend a mutable reference and an immutable reference at the same time.

```
fn main() {
    let mut v = vec![0u, 1, 2, 3];
    let last = &v[3];
    v.clear();
    println!("{}", last); // Dangling pointer?
}
```

Run

- References cannot outlive the scope of a value.

```
fn main() {
    println!("Your number was: {}", get_number());
}

fn get_number<'a>() -> &'a int {
    let a = 3;
    return &a;
}
```

Run

Mutability

- When can a resource be modified?

```
fn main() {
    let x = 42u;
    x = 43u;
}
```

Run

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

fn f(p: &Point) {
    p.y = 42;
}
fn main() {
    let p = Point { x: 0, y: 0 };
    f(&p);
}
```

Run

The best thing

- Not just annotations for the compiler.
- They **describe** the system.
- They **document** the system.

```
struct WebRequest {
    connection: postgres::Connection,
    store: &SessionStore
}
```

Run

- They **help me** reason about the system.
- The compiler catches bugs in the code.
- ...but also in my thinking.

Other features

Pattern matching

- Allows deep inspection of values.
- Allows behaviour conditional on the shape of data.

Run

```
#[deriving(Show)]
enum Direction {
    North,
    East,
    South,
    West
}

fn reverse(dir: Direction) -> Direction {
    match dir {
        North => South,
        East => West,
        South => North,
        West => East,
    }
}

fn main() {
    println!("You should head: {}", reverse(North));
}
```

Traits

Run

```
trait HasSize {
    fn size(&self) -> uint;
}
struct Square {
    side: uint
}
impl HasSize for Square {
    fn size(&self) -> uint {
        self.side * self.side
    }
}
struct Rectangle {
    width: uint,
    height: uint
}
impl HasSize for Rectangle {
    fn size(&self) -> uint {
        self.width * self.height
    }
}
fn print_size<T: HasSize>(v: T) {
    println!("The shape has size: {}", v.size());
}
fn main() {
```

#[derive]

- Automatically derive useful traits.

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

fn is_zero(p: &Point) -> bool {
    *p == Point { x: 0, y: 0 }
}
fn main() {
    assert!(!is_zero(&Point {
        x: 2, y: 4
    }));
}
```

Run

Macros

- Meta-programming with macros "by example"
- Allows introducing convenient simplifications of programs

Run

```
#![feature(macro_rules)]

macro_rules! assert_eq(
    ($given:expr, $expected:expr) => ({
        match (&($given), &($expected)) {
            (given_val, expected_val) => {
                if (*given_val != *expected_val) {
                    panic!("{} != {}", *given_val, *expected_val);
                }
            }
        }
    })
);

fn main() {
    assert_eq!(4u, 3u);
}
```

Parallelism

- Designed with *safe* parallelism in mind
  - Ownership semantics guarantee absence of data races
- Built-in basic support for SIMD

Run

```
use std::iter::MultiplicativeIterator;
use std::sync::Future;
use std::slice::SliceConcatExt;

fn factorial(x: uint) -> uint {
    std::iter::range_inclusive(1, x).product()
}

fn main() {
    println!("{}", range(0, 10)
        .map(|n| Future::spawn(move || factorial(n)))
        .map(|mut f| f.get().to_string())
        .collect::<Vec<_>>()
        .len());
}
```

Concurrency

- Message passing-based communication via channels

```
fn main() {
    let (tx, rx) = channel();
    spawn(proc () {
        tx.send(2u + 2);
        tx.send(3u + 5);
    });
    println!("{}, {}", rx.recv(), rx.recv());
}
```

Run

- ...but many other concurrency primitives exposed with safe interfaces

Iterators

- Lazy.
- A major abstraction in libraries.

Run

```
struct Town {
    population: uint,
    name: &'static str
}
fn show_most_populated_town_under_450k(towns: Vec<Town>) {
    println!("Most populated town under 450k is {}", towns.iter()
        .filter(|town| town.population < 450 * 1000)
        .max_by(|town| town.population)
        .map(|town| town.name));
}
fn main() {
    let towns = vec![
        Town { name: "Gothenburg", population: 491630 },
        Town { name: "Malmö", population: 278523 },
        Town { name: "Stockholm", population: 789024 },
        Town { name: "Karklskrona", population: 35212 },
        Town { name: "Lund", population: 82800 }
    ];
    show_most_populated_town_under_450k(towns);
}
```

- Compile down to very efficient code, usually equivalent to hand-rolled loops.

Built-in testing and benchmark harness

Run

```
use std::iter::MultiplicativeIterator;
fn factorial(n: uint) {
    range(1, n + 1).product()
}

#[test]
fn test_factorial() {
    assert_eq!(factorial(1), 1);
    assert_eq!(factorial(4), 24);
}

const BENCH_SIZE: uint = 20;
#[bench]
fn iterative_fibonacci(b: &mut Bencher) {
    b.iter(|| {
        fibonacci_sequence().take(BENCH_SIZE).collect::<Vec<uint>>(
    })
}
```

And if you really need it...

Unsafe

```
fn add_two_numbers_and_print(x: *const u64, y: *const u64) {
    println!("{} + {} = {}", *x, *y, *x + *y);
}

fn main() {
    let x = 1 as *const u64;
    let y = 2 as *const u64;
    add_two_numbers_and_print(x, y);
}
```

Run

And if you really need it...

Unsafe

```
fn add_two_numbers_and_print(x: *const u64, y: *const u64) {
    println!("{} + {} = {}", *x, *y, *x + *y);
}

fn main() {
    let x = 1 as *const u64;
    let y = 2 as *const u64;
    add_two_numbers_and_print(x, y);
}
```

Run

But you probably won't.

```
$ cd cargo
$ cat **/*.rs | wc -l
24700
$ git grep unsafe src | wc -l
0
```

```
$ cd rubigo-luculenta
$ cat **/*.rs | wc -l
2999
```

Servo

Research project to build the next-generation Web browser

Goals: security, mobile performance and power efficiency

Cargo package manager

- Reproducible builds.
- Central registry of crates and `cargo publish` coming soon.

```
[package]
name = "postgres"
version = "0.0.0"
authors = [ "Steven Fackler <sfackler@gmail.com>" ]

[lib]
name = "postgres"
path = "src/lib.rs"
test = false

[dependencies.openssl]
git = "https://github.com/sfackler/rust-openssl"

[dependencies.phf]
git = "https://github.com/sfackler/rust-phf"

[dependencies.phf_mac]
git = "https://github.com/sfackler/rust-phf"
```

Projects (known to be) using Rust

- Servo browser engine
- The Rust compiler
- Cargo package manager
- Piston game engine
- Zinc close-to-metal stack
- Iron Web framework
- Teepee
- Tilde
- OpenDNS

1.0 - what's being fleshed out

- Standard library stability
- ~~Dismantling of M:N concurrency~~
- ~~Moving out many built-in crates out to Cargo packages~~
- ~~Fleshing out terminology for spreading the good news~~
- Language features blocking the release:
  - Associated types
  - Unboxed clusores
  - ~~Polished macro system~~
- ~~Crate registry for Cargo (http://rubygems.org anyone?)~~

Rust 2 years ago and now

Run

```
enum Direction {
    North,
    East,
    South,
    West
}

fn main() {
    let mut table = ~[];
    for [0, ..10].each |i| {
        let mut row = ~[];
        for [0, ..128].each |j| {
            row.push(North);
        }
        table.push(copy row);
    }
    io::print(fmt!("before: %d\n", table[0][0] as int));
    setT(&mut table);
    io::print(fmt!("after: %d\n", table[0][0] as int));
}

fn setT(table: &mut ~[~[Direction]]) {
    for [0, ..10].each |i| {
        for [0, ..128].each |j| {
```

1.0 and stability

- Rust known for its backwards-incompatible changes in the 0.x cycle
- 1.0 will come with a *stability* guarantee
  - The core parts of the language will no longer change
  - Some parts of the standard library will no longer break
  - Feature gates will be used for non-stable parts of the language
  - A stable channel will only contain the stable parts of the language
  - A nightly channel will ship the unstable parts of the language
- The language will continue to develop
  - Rapid 6-week release cycle
  - http://blog.rust-lang.org/2014/09/15/Rust-1.0.html

Getting started

- **Amazing** community. User groups in:
  - California, London, Paris, Zurich, Malmö, Seattle, Toronto, Lisbon, Berlin, Tokyo, Milano and many more.
- 1230 Rust repositories on GitHub
- The Rust Book
- IRC: irc.mozilla.org #rust
- This Week in Rust
- http://github.com/rust-lang
- http://doc.rust-lang.org
- Playpen: http://play.rust-lang.org
- Rust By Example: http://rustbyexample.com
- Writing a path tracer in Rust: http://ruudvanasseldonk.com/

Questions?

jakub@jakub.cc

@JAW_dropping