Introduction to Rust

Ian McCormack
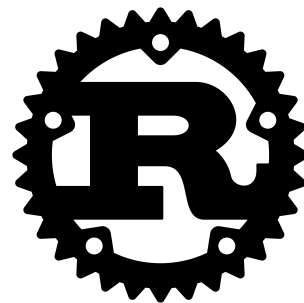
Software and Societal Systems Department
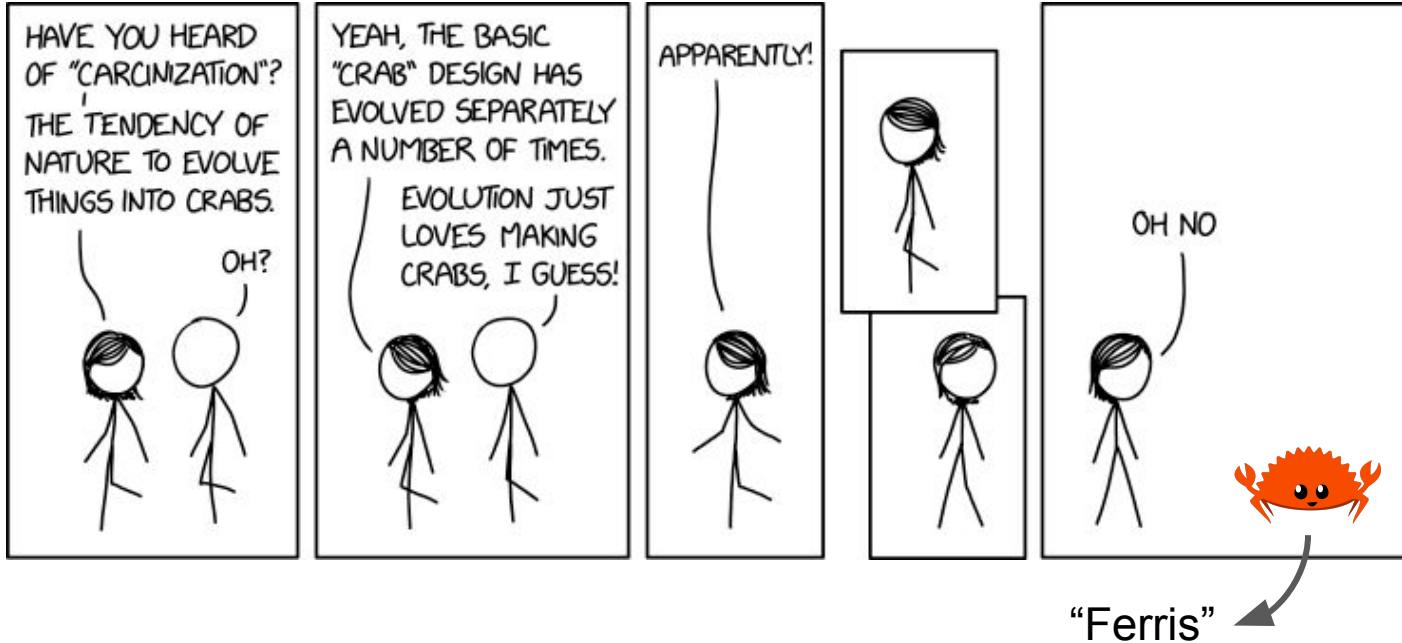
Jonathan Aldrich

Joshua Sunshine

Ian McCormack
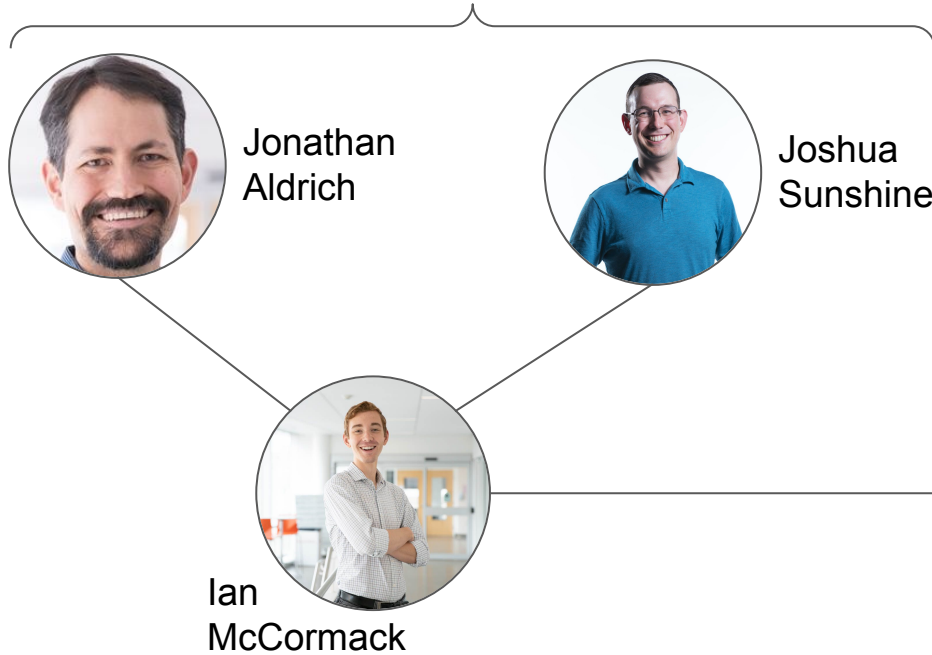
University of Illinois Urbana-Champaign
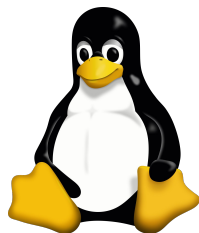
Timothy Zhou

Why should I use Rust?

Rust's mascot is a crab, and crabs are *awesome*.

"Ferris"

Rust is popular and widely used in production.

Chosen as **the "most loved" language** in the Stack Overflow's annual developer survey for **the last six years.**

Rust is energy, time, and memory efficient.

	Total							
	Energy (J)			Time (ms)			Mb
(c) C	1.00		(c) C	1.00		(c) Pascal	1.00
(c) Rust	1.03		(c) Rust	1.04		(c) Go	1.05
(c) C++	1.34		(c) C++	1.56		(c) C	1.17
(c) Ada	1.70		(c) Ada	1.85		(c) Fortran	1.24
(v) Java	1.98		(v) Java	1.89		(c) C++	1.34
(c) Pascal	2.14		(c) Chapel	2.14		(c) Ada	1.47
(c) Chapel	2.18		(c) Go	2.83		(c) Rust	1.54
(v) Lisp	2.27		(c) Pascal	3.02		(v) Lisp	1.92
(c) Ocaml	2.40		(c) Ocaml	3.09		(c) Haskell	2.45
(c) Fortran	2.52		(v) C#	3.14		(i) PHP	2.57
(c) Swift	2.79		(v) Lisp	3.40		(c) Swift	2.71
(c) Haskell	3.10		(c) Haskell	3.55		(i) Python	2.80
(v) C#	3.14		(c) Swift	4.20		(c) Ocaml	2.82
(c) Go	3.23		(c) Fortran	4.20		(v) C#	2.85
(i) Dart	3.83		(v) F#	6.30		(i) Hack	3.34
(v) F#	4.13		(i) JavaScript	6.52		(v) Racket	3.52
(i) JavaScript	4.45		(i) Dart	6.67		(i) Ruby	3.97
(v) Racket	7.91		(v) Racket	11.27		(c) Chapel	4.00
(i) TypeScript	21.50		(i) Hack	26.99		(v) F#	4.25
(i) Hack	24.02		(i) PHP	27.64		(i) JavaScript	4.59
(i) PHP	29.30		(v) Erlang	36.71		(i) TypeScript	4.69
(v) Erlang	42.23		(i) Jruby	43.44		(v) Java	6.01
(i) Lua	45.98		(i) TypeScript	46.20		(i) Perl	6.62
(i) Jruby	46.54		(i) Ruby	59.34		(i) Lua	6.72
(i) Ruby	69.91		(i) Perl	65.79		(v) Erlang	7.20
(i) Python	75.88		(i) Python	71.90		(i) Dart	8.64
(i) Perl	79.58		(i) Lua	82.91		(i) Jruby	19.84

What is **memory safety**?

Rust's main selling point is strong **static memory safety.**

Garbage collection supports **dynamic memory safety**.

Tracing garbage collection treats memory as a reachability graph, and periodically eliminates nodes that are unreachable.

Reference counting is continuous; objects with no referents are freed.

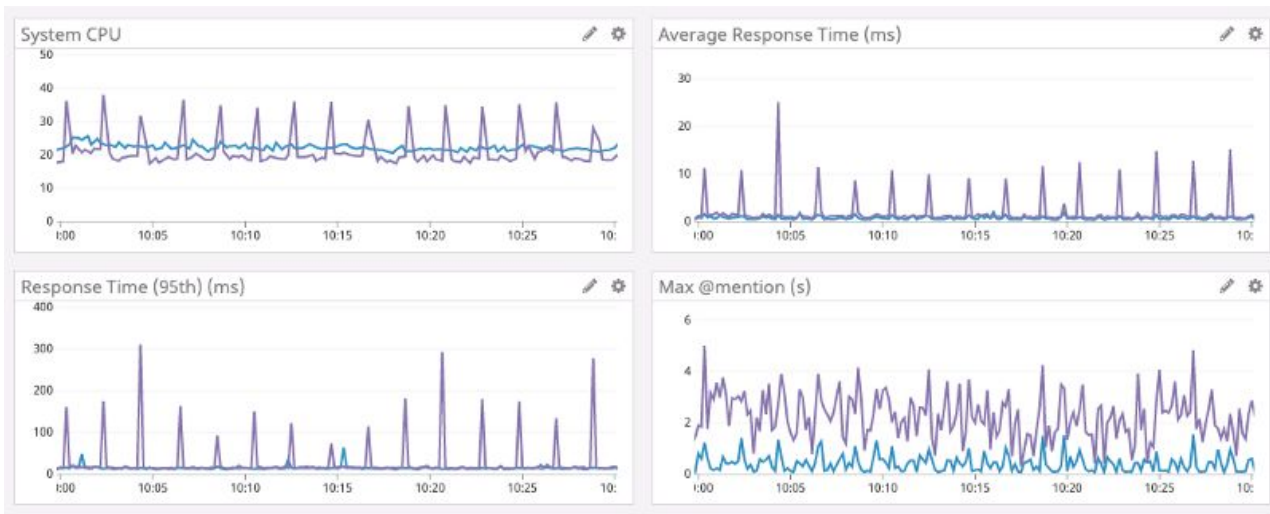**Java** and **Go** use **tracing** garbage collection.

Check out *Crafting Interpreters*!

CRAFTING
INTERPRETERS
ROBERT NYSTROM
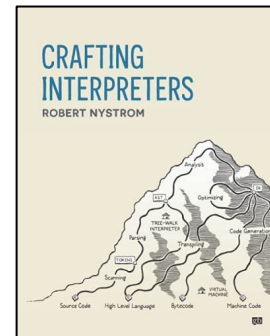
Garbage collection causes runtime overhead.

Performance of Discord's Read States Service

Go

Rust

Rust has similar *performance*, *energy efficiency*, and *memory usage* to C, with **static memory safety** guarantees.

Rust

- Basic language features
- Ownership & the borrow checker
- Generic types

Rust Resources

The Rust Programming Language — https://doc.rust-lang.org/book/

A tutorial on every aspect of Rust; a great starting point.

The Rustonomicon — https://doc.rust-lang.org/nomicon/

Explains advanced features, such as 'unsafe' and foreign function use.

Basic language features:
Why should we talk about this?

Mutability

Variables are immutable by default.

Can be declared as mutable with the 'mut' keyword.

```
let x = 5;
x = 7;
```

```
let mut x = 5;
x = 7;
```

Why do we have
immutability by default?

Numerical Types

Numerical Types

Length	Signed	Unsigned
8-bit	`i8`	`u8`
16-bit	`i16`	`u16`
32-bit	`i32`	`u32`
64-bit	`i64`	`u64`
128-bit	`i128`	`u128`
arch	`isize`	`usize`

Numerical Literals

Number literals	Example
Decimal	`98_222`
Hex	`0xff`
Octal	`0o77`
Binary	`0b1111_0000`
Byte (`u8` only)	`b'A'`

```
let mut x:u16 = 0xffff;
```

Shadowing

```
let x:u16 = 0xffff;
{
    let x:u32 = 0xffffffff;
    {
        let x:u64 = 0xffffffffffffffff;
    }
}
```

```
let x:u16 = 0xffff;
let x:u32 = 0xffffffff;
let x:u64 = 0xffffffffffffffff;
```

When would this be
useful to have?

Tuples & Arrays

Tuples

```
let x:(i32, &str, char) = (1, "hello", '@');

let (i, s, c) = x;

let y = x.2;
```

Arrays

```
let x:[i32;3] = [1, 1, 1];

let y = [1;3];              // also [1, 1, 1]

let slice = &x[0..1];      //        [1, 1]
```

Statements vs. Expressions

Expressions evaluate to a value; statements do not.

Semicolons distinguish between an expression and a statement.

```
let x = let y = let z = 5;
```

```
let x = {
    let z = -b + sqrt(pow(b,2) - 4 * a * c);
    z / 2 * a
}
```

Conditional Statements

```
let x:i32 = if( cond ) {
    ...
}else{
    ...
}
```

Conditionals can be used as expressions.

A condition must be of type 'bool'; Rust doesn't convert types to 'bool'.

```
let x:i32 = if( y = 5 ) {
    ...
}else{
    ...
}
```

?

Loops

```
let t = loop {
    break 5;
};
```

```
use nix::unistd::{fork};

while(true){
    fork();
}
```

```
for i in (0..100) {
    println!(i);
}
```

Functions

```
fn foo(num:u32) -> u32 {
    ...
    let ex_closure = |num:u32| -> u32 {
        ...
    };

    num + 5
}
```

Enums & Matching

```
match something_error_prone() {

    Ok(success_value) => { ... }

    Err(err_value) => { ... }

}
```

```
match optional_result() {

    Some(value) => { ... }

    None => { ... }

}
```

Structs & Traits

```
struct Ian {
    pub publications: Vec<Publication>
}

pub trait GradStudent {
    fn submit_publication(&self) -> Vec<Result<Accept, Reject>>
}

impl GradStudent for Ian {

    fn submit_publication(&self) -> Vec<Result<Accept, Reject>>{
        self.publications
            .map(|_| Reject)
            .collect::<Vec<Result<Accept, Reject>>>()
    }
}
```

The Borrow Checker

Rust's Borrow Checker does the following:

- That you can't **move** the same value twice.

- That you can't **move** a value while it is **borrowed**.

- That you can't access a place while it is **mutably borrowed** (except through the reference).

- That you can't mutate a place while it is **immutably borrowed**.
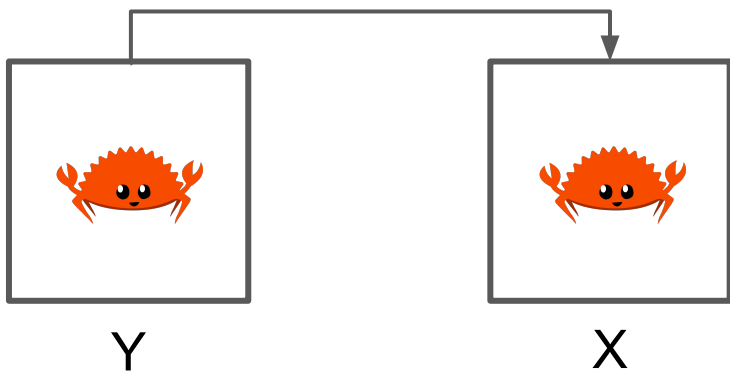
From *"Guide to Rustc Development"*

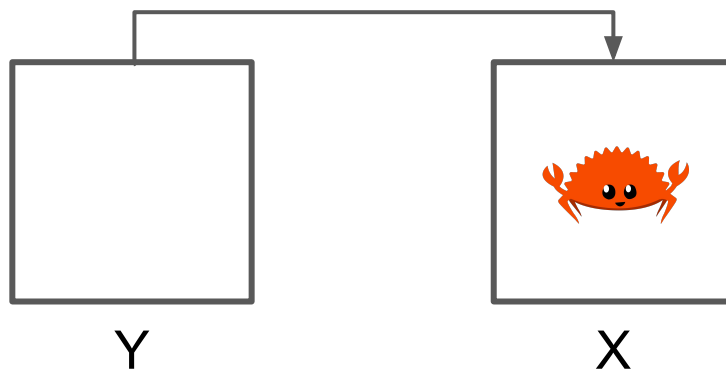# Assignment behaves differently for different types.

Copy Semantics

Move Semantics

$x = y$

$x = y$

Y   X

Y   X

What's the point of this?

Strings have move semantics.

```
fn conversation(){
    let greeting = String::from("Hello world!");

    let response = greeting;
    response.pop();
    response.insert(response.len(), '?');

    println!("{}", greeting);
    println!("{}", response);
}
```

Strings have move semantics.

```
fn conversation(){
    let greeting = String::from("Hello world!");

    let response = greeting;
    response.pop();
    response.insert(response.len(), '?');

    println!("{}", greeting);
    println!("{}", response);
}
```

We can avoid moves with borrowing.

```
fn conversation(){

    let greeting = String::from("Hello world!");        // String

    let response = &greeting[0..11];                     // &str

    println!("{}", greeting);

    println!("{}?", response);
}
```

We can avoid moves with borrowing.

```
fn conversation(){

    let greeting = String::from("Hello world!");

    let response = &greeting[0..11];

    println!("{}", greeting);

                        greeting.clear();            // greeting == ""

    println!("{}?", response);
}
```

We can avoid moves with borrowing.

```
error[E0502]: cannot borrow `greeting` as mutable because it is also
borrowed as immutable
 --> src/lib.rs:8:2
   |
4  |       let response = &greeting[0..11];                    // &str
   |                      --------- immutable borrow occurs here
...
8  |       greeting.clear();
   |       ^^^^^^^^^^^^^^^^ mutable borrow occurs here
9  |
10 |       println!("{}?", response);
   |                       -------- immutable borrow later used here
```

Rust has awesome error messages!

What does it mean to borrow something as mutable?

```rust
fn conversation(){

    let greeting = String::from("Hello world!");

    let immut_greeting = & greeting;        // immutable borrow

    let mut greeting = & mut greeting;      // mutable borrow


    (*immut_greeting).clear();              // greeting == ""

    (*mut_greeting).clear();
}
```

Rust's borrow checker is...

"…a system for statically building a proof that data in memory is either uniquely owned (and thus able to allow unguarded mutation) or collectively shared, but not both."

Check out *Oxide: the Essence of Rust*!

Mutability

Variables are immutable by default.

Can be declared as mutable with the 'mut' keyword.

```
let x = 5;
x = 7;
```

```
let mut x = 5;
x = 7;
```

Why do we have
immutability by default?
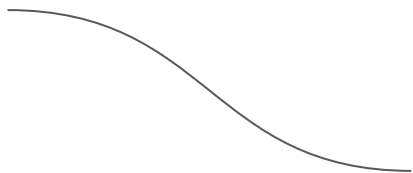
A reference is one of two things:

1. Unique and mutable
2. Sharable and immutable

How does Rust verify this?

The borrow checker reasons using **lifetimes**.

The **scope** of a value is the duration for which it is **allocated.**

The **lifetime** of a reference is the duration for which it is **used.**

Must form a partial ordering.

v ⌐ &v ⌐ &&v ⌐ …

What are the lifetimes and scopes here?

```
fn conversation(){

    let greeting = String::from("Hello world!");

    let response = &greeting[0..11];

    println!("{}", greeting);

    greeting.clear(); == call_function(& mut greeting);

    println!("{}?", response);
}
```

```
error[E0502]: cannot borrow `greeting` as mutable because it is also
borrowed as immutable
  --> src/lib.rs:8:2
   |
4  |         let response = &greeting[0..11];                      // &str
   |                         -------- immutable borrow occurs here
...
8  |         greeting.clear();
   |         ^^^^^^^^^^^^^^^^^ mutable borrow occurs here
9  |
10 |         println!("{}?", response);
   |                         -------- immutable borrow later used here
```

Rust restricts assignments based on lifetimes.

```
fn assign<'a, 'b>(mut a: & 'a i32, b: & 'b i32)
    where 'b:'a
{
    a = b;
}
```

?

```
error: lifetime may not live long enough
 --> src/lib.rs:2:5
  |
1 |  fn assign(mut a: & i32, b : & i32){
  |                   -       - let's call the lifetime of this reference`'1`
  |                           |
  |                           let's call the lifetime of this reference `'2`
2 |     a = b;
  |     ^^^^^ assignment requires that `'1` must outlive `'2`
```

Lifetimes are *tricky*.

1. Defined differently by different formalisms.
2. Changed throughout the development of Rust.
3. Require type annotations.

Lifetimes are types.

```java
class Animal { ... }

class Crab extends Animal { ... }

public static void main(String[] args){

    Crab c = (Crab) new Animal(); X
    Animal a = (Animal) new Crab();

}
```

Writing a value of a longer lifetime to a shorter lifetime?

```
Animal a = (Animal) new Crab();
```

Writing a value of a shorter lifetime to a shorter lifetime?

```
Crab a = (Crab) some_animal;        // some_animal instanceof Animal
```

Exercise:

Complete the signature and implementation for the following versions of the
'swap' function without using any other dependencies. Some signatures aren't
entirely complete.

1. fn swap(x: & i32, y: & i32)

2. fn swap(x: & & i32, y: & & i32)

3. fn swap<**T**>(x: & **T**, y: & **T**)

Each should swap the values of the two parameters such that, after the function
returns, we have:

*y = old(*x) && *x = old(*y)

https://play.rust-lang.org/
https://tinyurl.com/3y3wz4sz

Answer #1

```
fn swap<T>(x: & mut i32, y: & mut i32)
{
    let temp = *x;
    *x = *y;
    *y = temp;
}
```

Answer #2

```
fn swap<T, 'a>(x: & mut 'a & T, y: & mut 'a & T)
{
    let temp = *x;
    *x = *y;
    *y = temp;
}
```

Answer #3

```
fn swap<T>(x: & mut T, y: & mut T)                        std::mem::swap
{
    unsafe {
        let a = ptr::read(x);
        let b = ptr::read(y);
        ptr::write(x, b);
        ptr::write(y, a);
    }
}
```

Research Overview

Memory safe languages aren't *truly* safe.

CVE-2022-37454 — A buffer overflow vulnerability in the *reference implementation* of the SHA-3 hash function for Python and PHP.

```python
import hashlib
h = hashlib.sha3_224()

h.update(b"\x00" * 1)
h.update(b"\x00" * 4294967295)

print(h.hexdigest())
```

Pushed in 2011, CVE disclosed last month.

Rust's standard library uses unsafe for "axiomatic"[1] operations.

```
1:   fn swap<T>(x: &mut T, y: &mut T)                      std::mem::swap
2:   {
3:       unsafe {
4:           let a = ptr::read(x);
5:           let b = ptr::read(y);
6:           ptr::write(x, b);
7:           ptr::write(y, a);
8:       }
9:   }
```

[1]Scott, 2019

Calling Foreign Functions

```
1:   extern "C" {                                    Example Code
2:       int foo(*mut i32);
3:   }
4:
5:   int main() {
6:       let x = 5;
6:       unsafe {
7:           return foo(ptr::read(&x));
8:       }
9:   }
```

Félix
@fayfiftynine

Proposal: memory-safe languages should stop using the word "unsafe" for escape hatches because using them makes you feel like a cool pro who can handle dangerous things. I propose "hubris"

1:24 PM · 9 Aug, 2022

56 replies   **151** shares   **1K** likes

52

Rust's most popular package repository is **crates.io** 📦

Rust refer to **'package'** or **'library'** as a **'crate'**.

95,620 crates published as of 10/30.

Categories of Unsafe Function Calls

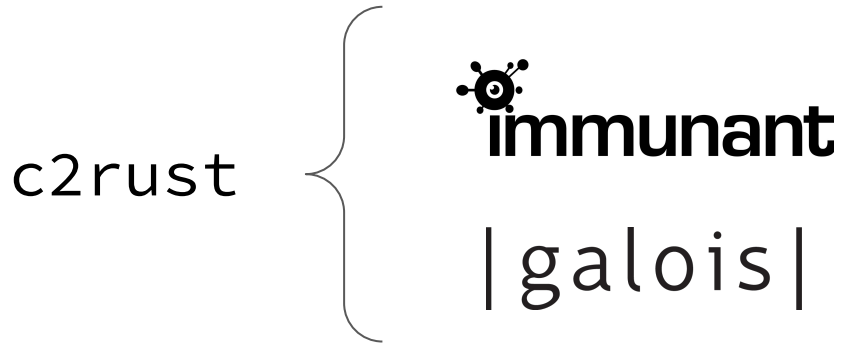**47.2%**

Rust's standard library

36.4% C-style pointer use

30.3%

User-written or 3rd-party

22.5%

Foreign Functions

[1]Evans et al., ICSE 2020

Automatically translating C to Rust

c2rust

immunant

|galois|

Existing C Code	→	Unsafe Rust Code	→	Safe Rust Code

Categories of unsafety observed in 17 C repositories.

RawDeref - dereferencing a raw pointer

Global - read/write global state

Union - read a field from a C-style union.
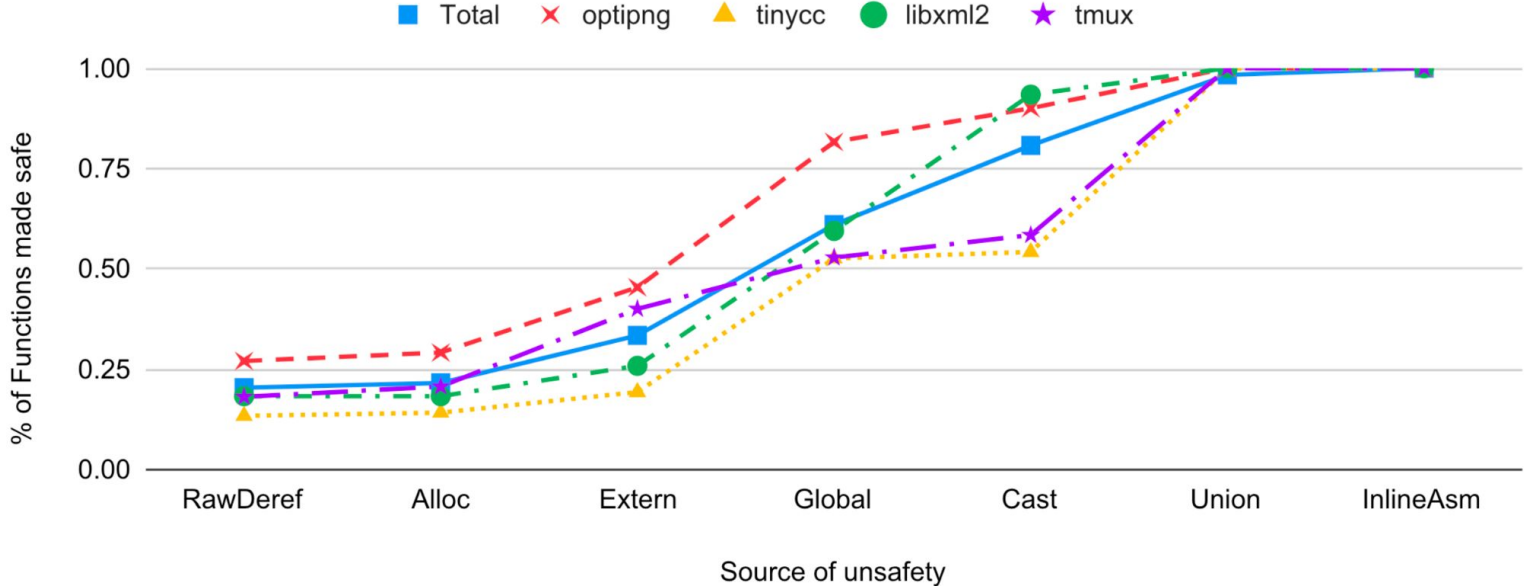
Allocation - direct calls to alloc/free

Extern - calling foreign functions or function pointers

Cast - unsafe casting

InlineASM - inline assembly instructions

[1]Emre et al., OOPSLA 2021

Unsafe features are close in proximity.

[1]Emre et al., OOPSLA 2021

Source-to-source transformation is a possible solution.

Using transformations written in the **TXL** language, achieved > 95% reduction in unsafe functions.

Project name	LOC	# func	c2rust	CRustS
TLVcodec[7]	136092	73	0.014	**0.959**
tinycc v0.9.27	72394	527	0.002	**0.991**
RosettaCode*	7328	381	0.223	**0.995**
Checked_C	724402	3535	0.002	**0.997**
ptrdist-1.1 [8]	9857	236	0.021	**0.983**
BusyBox	303275	4589	0.003	**0.647**

RosettaCode*: summed and averaged over 85 tasks in RosettaCode

Soundness of program transformation.

To convert unsafe *pointers* to safe *references,* we need an analysis that:

Can prove ownership

Can infer lifetime information

Rust Spec

Emre et al. use the Rust compiler as an oracle.

How does this problem evolve if we don't target *rewriting* C?

Lifetime Inference for C/C++

Instead of translating C/C++ to Rust, infer wrappers around unsafe foreign function calls.

```
fn example<'a, 'b, T>(x: & mut & 'a T, y: & mut & 'b T)
    where 'a:'b, 'b:'a
{

    unsafe {

        …
        let result **i32 = some_function_in_c(x, y)

    }

    ...
}
```