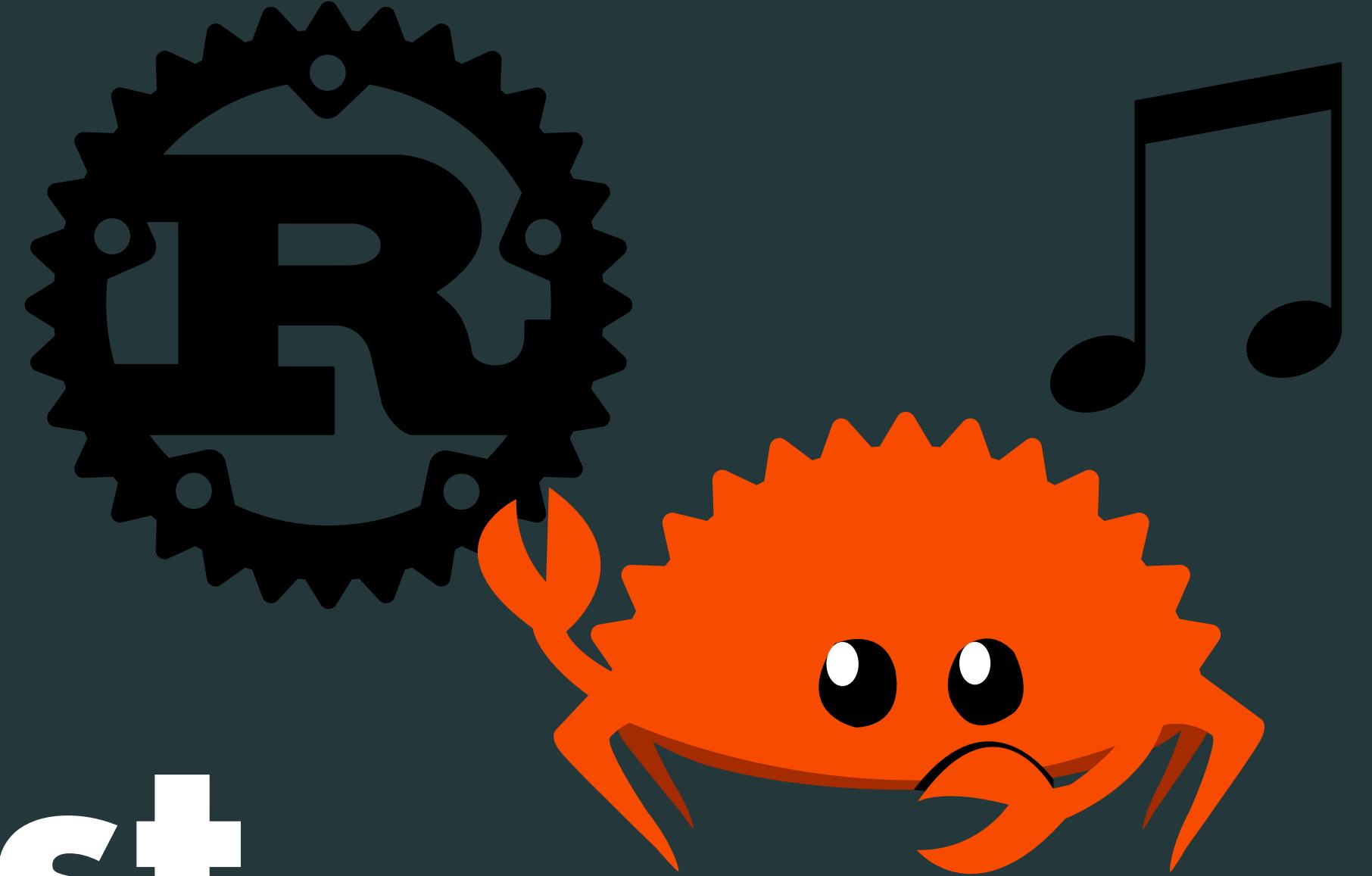
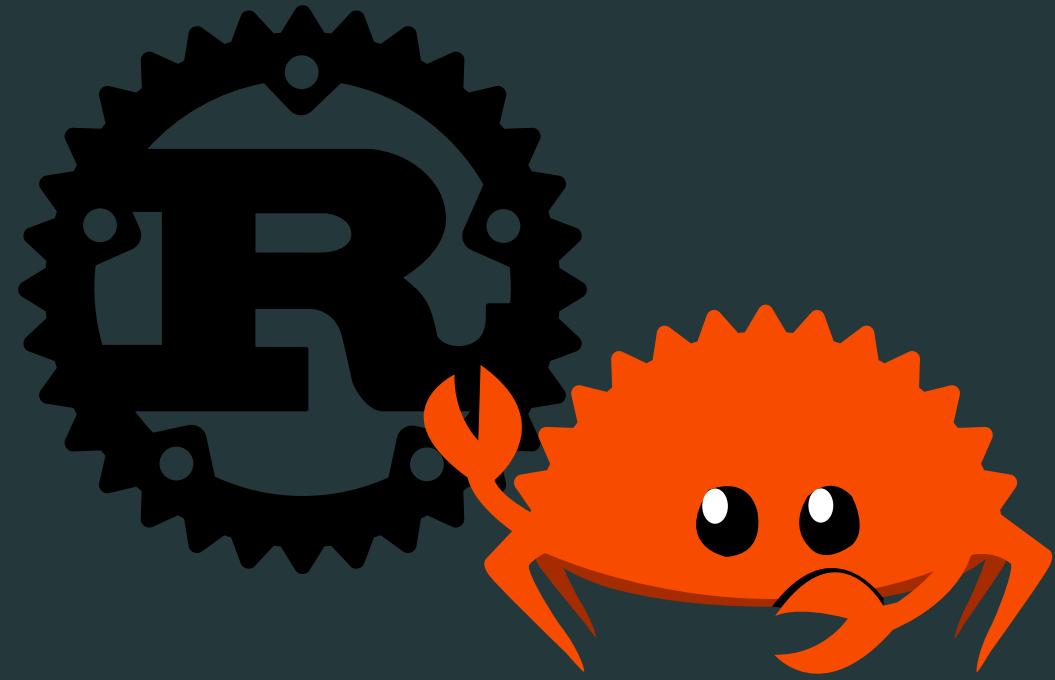


The Fun of RUST

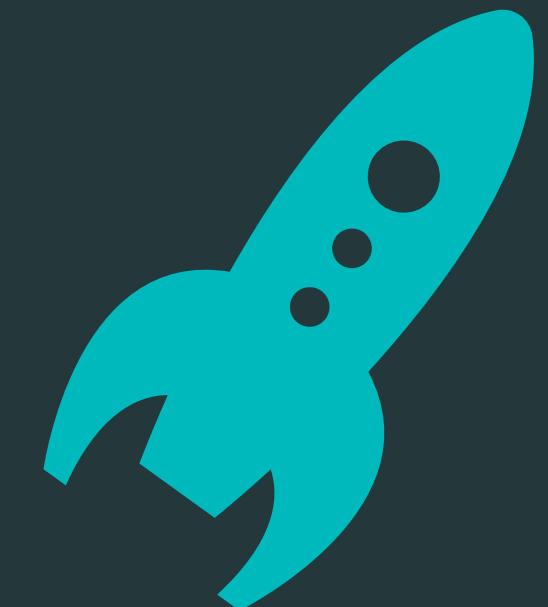
MATSUSHITA Yusuke – Igarashi & Suenaga Lab

Dec 20, 2024 – Talk for the Course of CCE, GS of Informatics, Kyoto University





Rust Takes Us to the New Age of Software Development

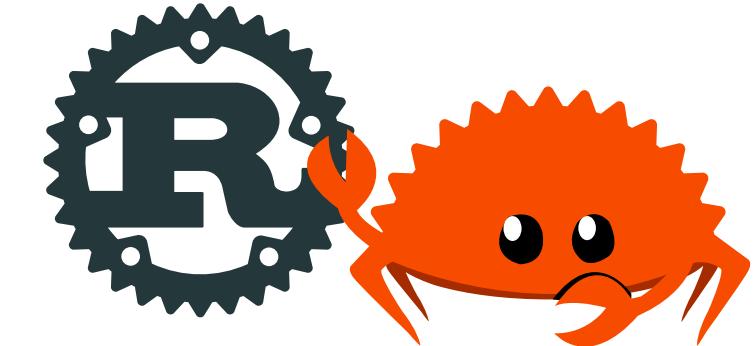


About Me



At ACM PLDI 2022

MATSUSHITA Yusuke

- ◆ **Software scientist** 
 - ▶ Solid theories for real-world practice
- ◆ **Loves & studies Rust** 
 - ▶ Rust is fun
- ◆ **Loves music** 
 - ▶ Esp. improvisation

More about Me



Lecturer at IPA Security Camp 2024
S15 Rust Program Verification Seminar



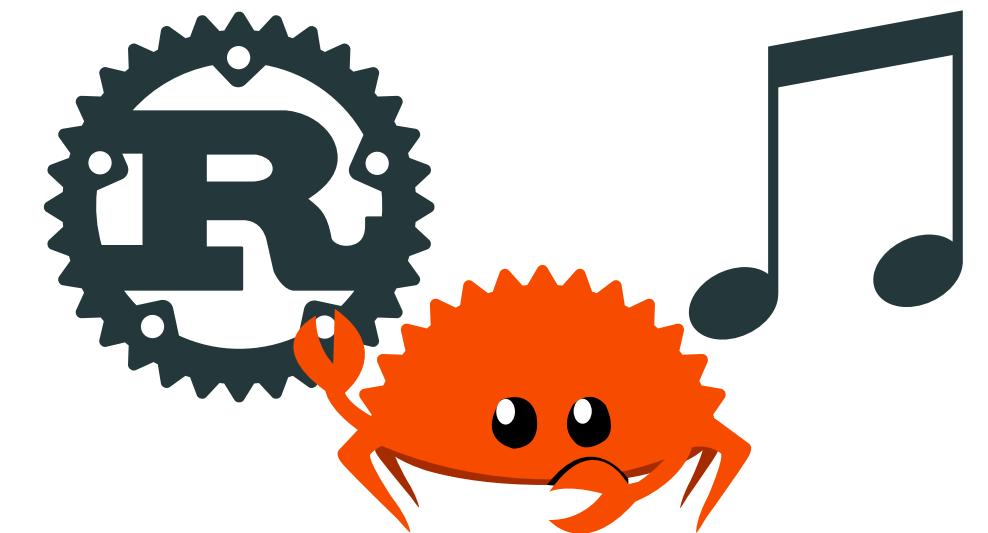
Got a Ph.D. in 2024 at
the Dept. of Computer Science,
GS of IST, the University of Tokyo
Supervised by Prof. Naoki Kobayashi



Japan Bach Concours 2022
Gold Prize
Ricercar a 3, the Musical Offering

This Talk

- ♦ Tells you the fun of Rust
 - ▶ Esp. regarding ownership types & formal verification



Outline

- #0 Software Science #1 The Rust Language
- #2 Rust's Ownership Types #3 Formal Verification for Rust

Questions for you

Why do so many software engineers love Rust?

What is Rust in light of history? How will Rust change the future?

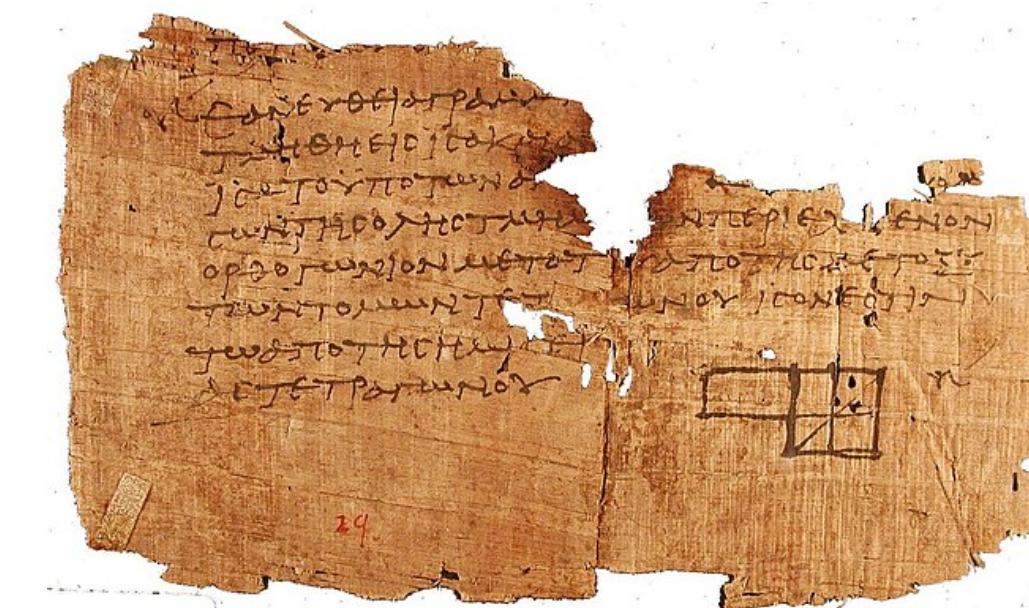
#0

Software Science

Ancient History of Computation

♦ ~300 BC Algorithm

- ▶ Euclidean algorithm, in Ancient Greece
 - For computing GCDs



Euclid's «Elements»

♦ ~150 BC Analog computer

- ▶ Antikythera mechanism, in Ancient Greece
 - Can compute astronomical positions and eclipses, allegedly



Marsyas, CC BY 2.5

Antikythera mechanism

Modern History of Computation

♦ 1804 Programming

- Jacquard Loom, in the industrial revolution
 - Punched cards for complex weaving patterns

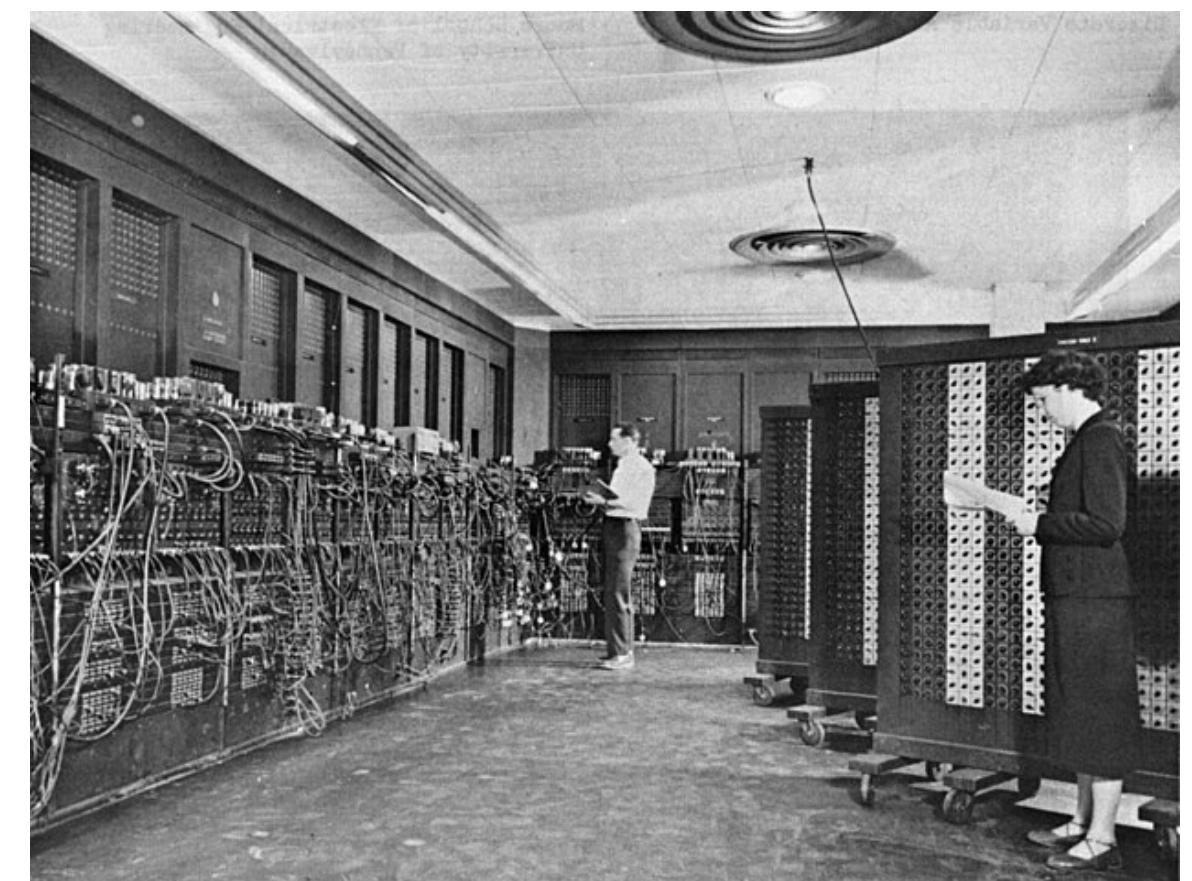


Stephencdickson, CC BY-SA 4.0

Jacquard Loom

♦ 1945 Digital computer

- ENIAC, towards the end of WWII
 - Financed by the US Army
 - Used for research on hydrogen bombs



ENIAC

Software & Bugs

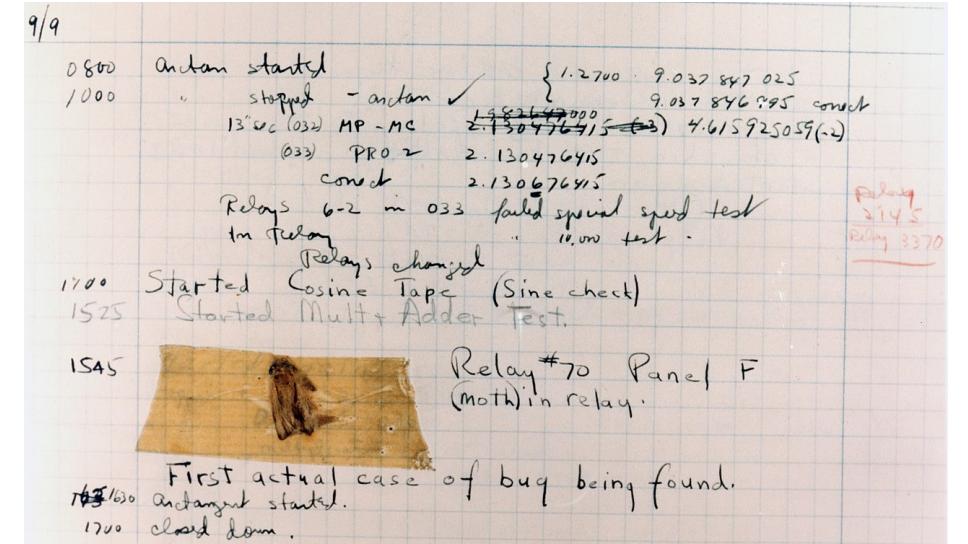
- ◆ **Software = Complex of programs, specs, etc.**

- ▶ Instructs computers to achieve high-level goals

- ◆ **Bug = Design error in software**

- ◆ **Bugs can cost**

- ▶ Example: NASA's Venus flyby Mariner I in 1962
 - Launch failure due to a spec bug, ~\$200M loss



“Bug” in a computer

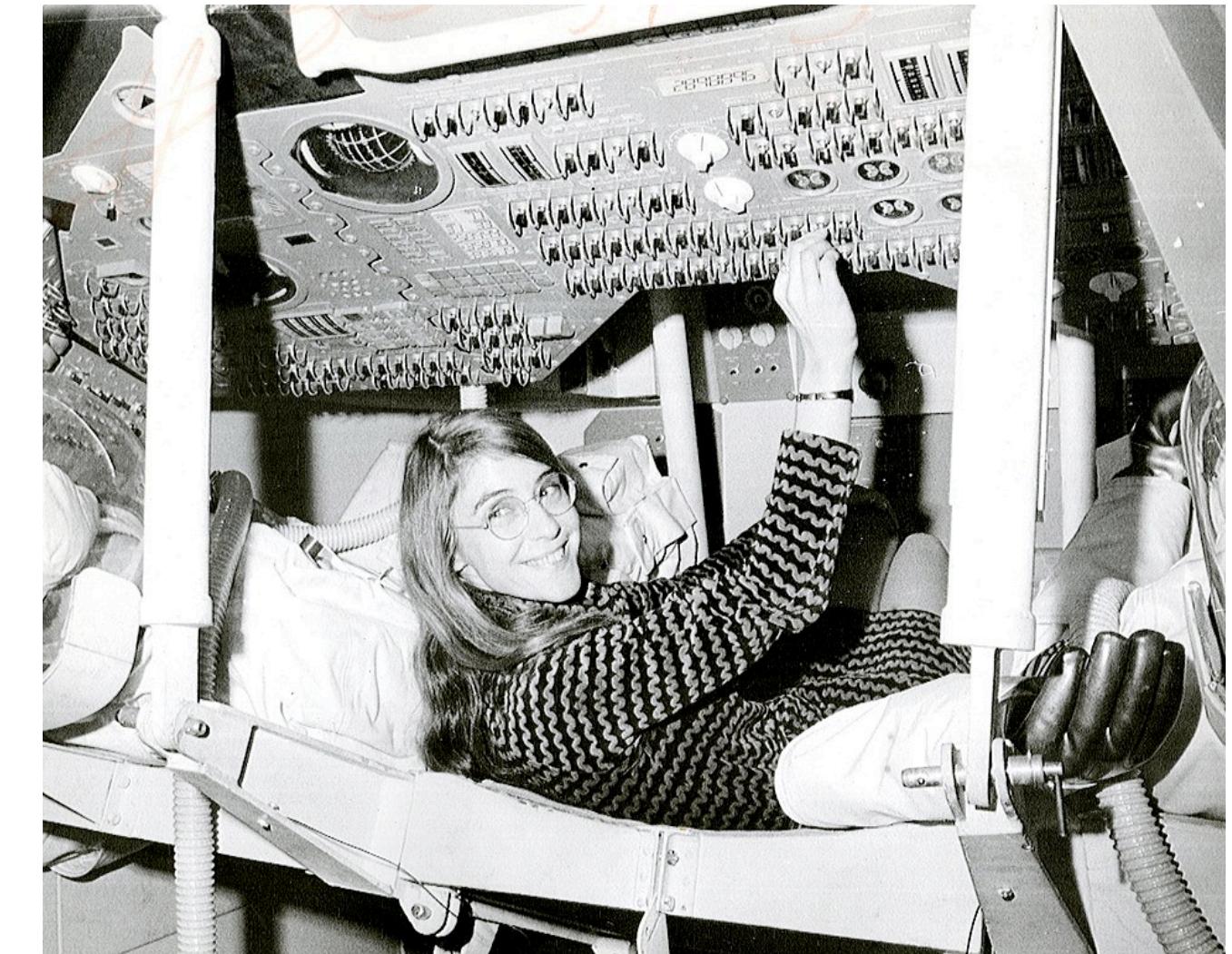


Launch of Mariner I

Software Engineering & Science

- ◆ Solid methodologies for building high-quality software
 - Margaret Hamilton, who led the development of software for NASA's Apollo 11 in the 1960s, advocated the name “software engineering”

- ◆ Core goal: Avoid critical bugs
 - Approaches: Programming languages, Types, Software testing & verification, Fail-safety, ...



Margaret Hamilton

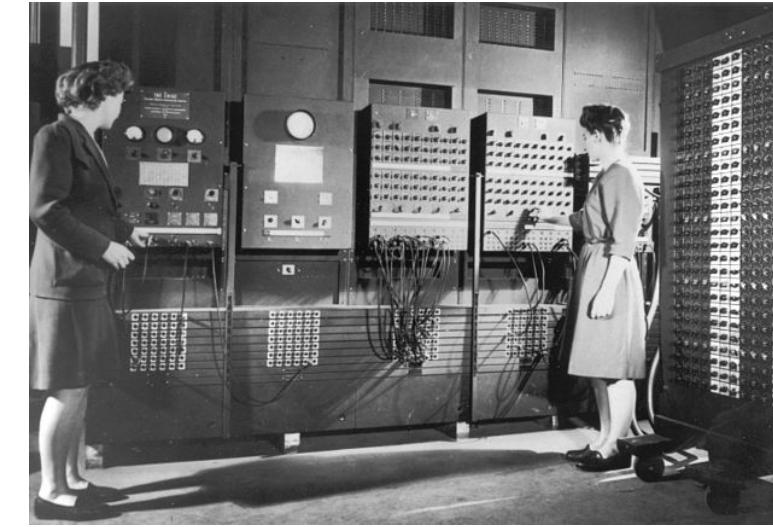


NASA has a team for studying formal methods

History of Programming

♦ 1940's Physical settings

- ▶ Plugboard wiring, switches, etc.



Programmers of ENIAC

♦ 1947 Assembly languages

- ▶ Wrapper of machine code

```
0013      RESETA EQU    %00010011
0011      CTLREG EQU    %00010001
C003 86 13 INITA   LDA A  #RESETA
C005 B7 80 04 STA A  ACIA
C008 86 11 LDA A  #CTLREG
C00A B7 80 04 STA A  ACIA
C00D 7E C0 F1 JMP    SIGNON
```

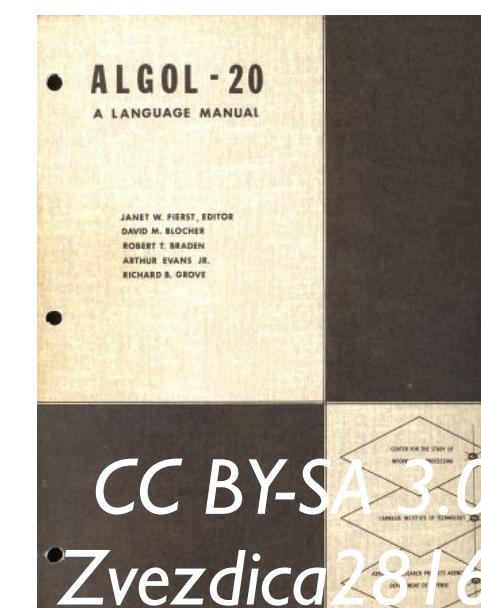
Assembly language

♦ 1952 Programming languages

- ▶ High-level abstraction
- ▶ Various designs have been explored

1958

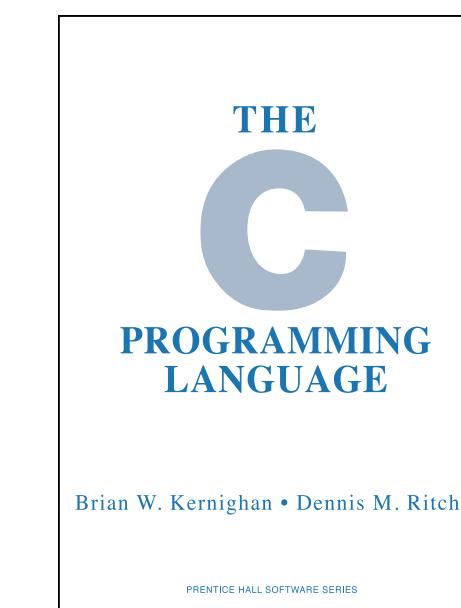
ALGOL



CC BY-SA 3.0
Zvezdica 28/16

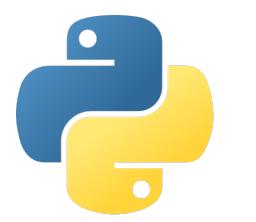
1972

C



1991

Python



1995

Java



2009

Go



and so on ...

Types in Programming Languages

♦ Labels that describe kinds, attributes, etc.

- ▶ A systematic way to prevent bugs in programs
- ▶ Static, compile-time vs. Dynamic, run-time
- ▶ Key topic in software engineering & science

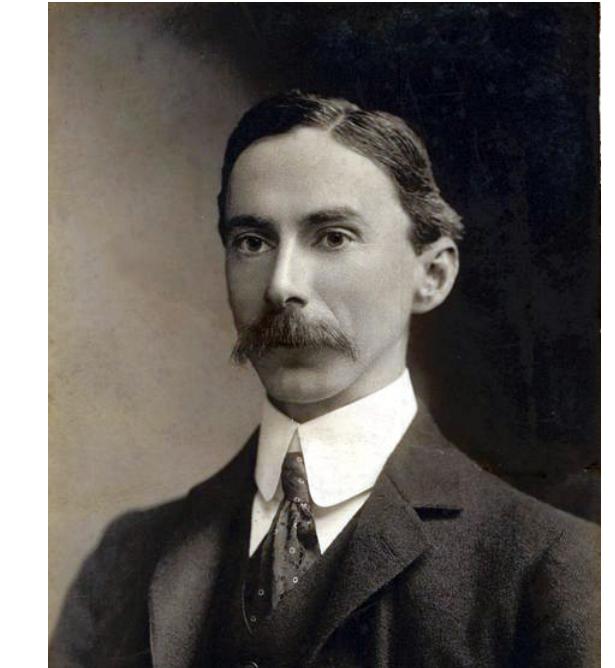
123 : Int

"Hello" : String



History of Types

- ◆ 1903 Russel's work on types for logic
 - ▶ To avoid Russel's paradox, caused by the set $\{S \mid S \notin S\}$
- ◆ 1940 Church's simply typed lambda calculus
 - ▶ Types for a well-behaved formal model of computation
- ◆ 1958 Types for programming languages
 - ▶ For preventing bugs in software development



Bertrand Russell



Alonzo Church

1. Type declarations

Type declarations serve to declare certain variables, or functions, to represent quantities of a given class, such as the class of integers or class of Boolean values.

Form: $\Delta \sim type(I, I, \dots, I, I[], \dots, I[,], \dots, I[,,], \dots)$ where *type* is a symbolic representative of some *type* declarator such as *integer* or *boolean*, and the *I* are identifiers. Throughout the program, the variables, or functions named by the identifiers *I*, are constrained to refer only to quantities of the type indicated by the declarator. On the other hand, all variables, or functions which are to represent other than arbitrary real numbers must be so declared.

Algol 58 Report, CACM

Types Can Be Really Rich

- ◆ **Types can tell value information**

- ▶ E.g., `{a:{b:0,c:7}} : {a:{b:0,c:number}}`

2012

TypeScript



- ◆ **Types can tell side effects**

- ▶ E.g., `getLine : IO String` (c.f. `"Hi" : String`)

1990

Haskell



- ◆ **Types can be propositions**

- ▶ E.g., `add_comm : ∀ m n, m + n = n + m`
- ▶ Curry-Howard correspondence

1989
Coq



Formal Verification

♦ Proving that the software satisfies the specs

- ▶ A rigorous way to eradicate software bugs
- ▶ Practically used in critical domains
 - Esp. in aerospace, systems & financial software

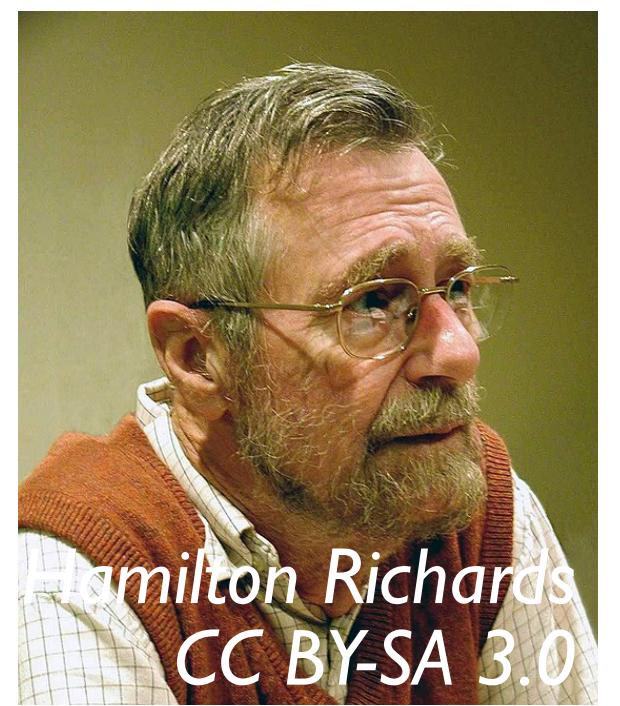


♦ A central topic in software science

- ▶ Program logics, especially Hoare Logic (1969)
 - Dijkstra's weakest precondition calculus (1975)
- ▶ Advanced types can also be used



Tony Hoare



Edsger Dijkstra

#1

The Rust Language

The Rust Language Is Hot



♦ Rust is performative, reliable, and productive

- ▶ Low-level memory & thread controls like C/C++
- ▶ High-level memory & thread safety by innovative ownership types
- ▶ Modern language features & ecosystem

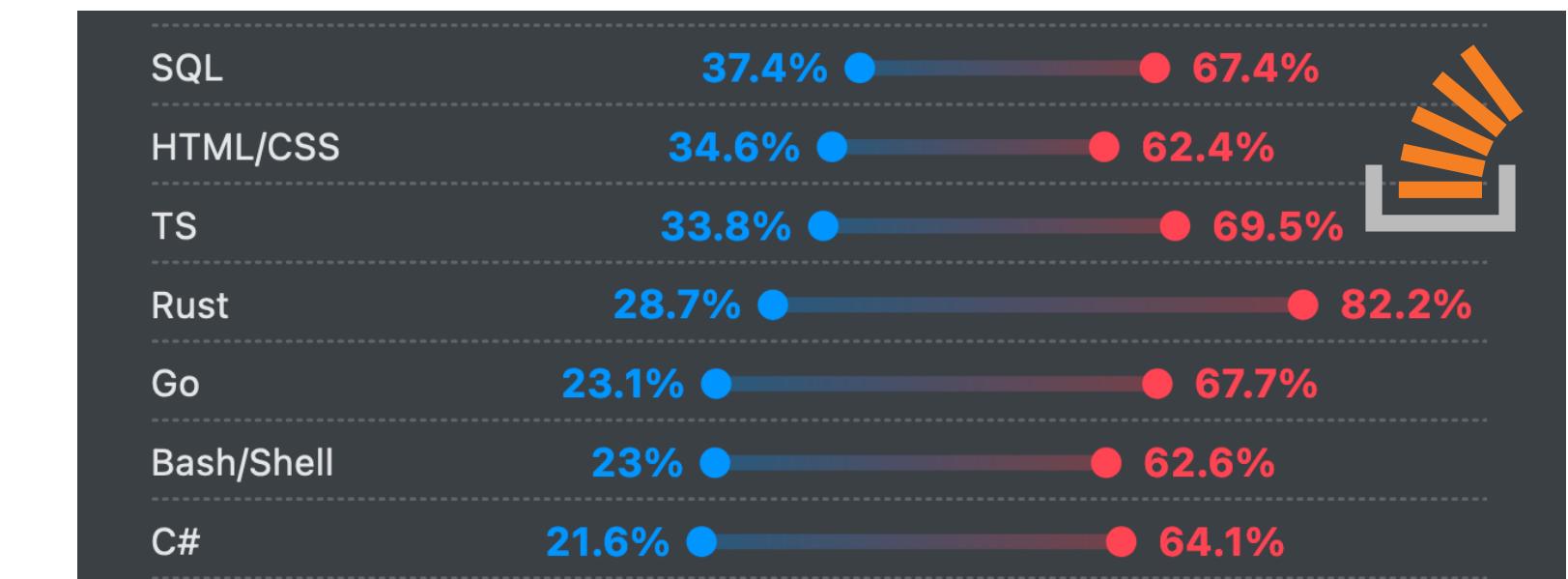
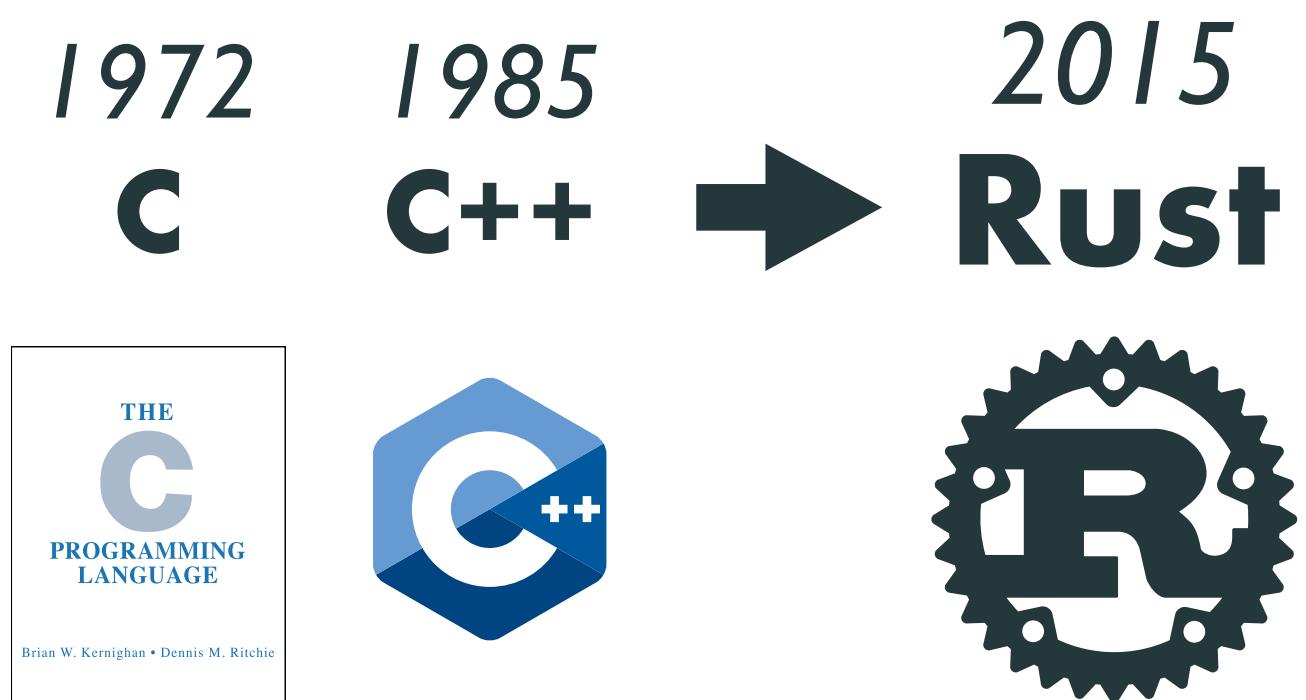
Rust

A language empowering everyone to build reliable and efficient software.

Why Rust?

Performance	Reliability	Productivity
Rust is blazingly fast and memory-efficient: with no runtime or garbage collector, it can power performance-critical services, run on embedded devices, and easily integrate with other languages.	Rust's rich type system and ownership model guarantee memory-safety and thread-safety – enabling you to eliminate many classes of bugs at compile-time.	Rust has great documentation, a friendly compiler with useful error messages, and top-notch tooling – an integrated package manager and build tool, smart multi-editor support with auto-completion and type inspections, an auto-formatter, and more.

GET STARTED
Version 1.83.0



Rust has been the most admired language for 9 years

Why Memory & Thread Safety?

♦ Memory causes critical bugs

- ▶ Use after free, buffer overflows, ...
- ▶ OpenSSL's Heartbleed bug in 2012 caused buffer over-reads
- ▶ ~70% of the zero-day attacks were due to memory corruption (Google Zero 2019)

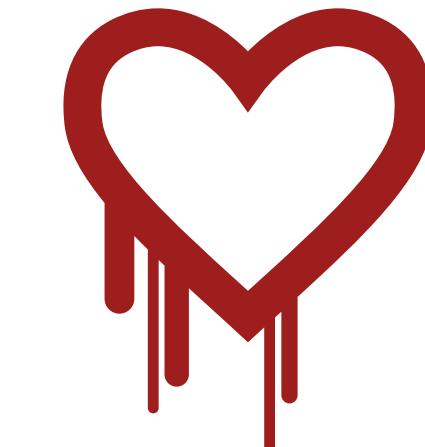
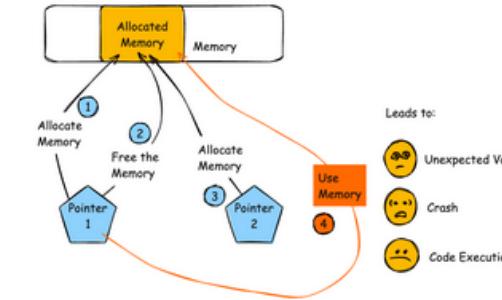
CWE-416: Use After Free

Weakness ID: 416
Vulnerability Mapping: ALLOWED
Abstraction: Variant

View customized information: Conceptual Operational Mapping Friendly Complete Custom

Description

The product reuses or references memory after it has been freed. At some point afterward, the memory may be allocated again and saved in another pointer, while the original pointer references a location somewhere within the new allocation. Any operations using the original pointer are no longer valid because the memory "belongs" to the code that operates on the new pointer.



Heartbleed bug

♦ Multithreading is super tricky

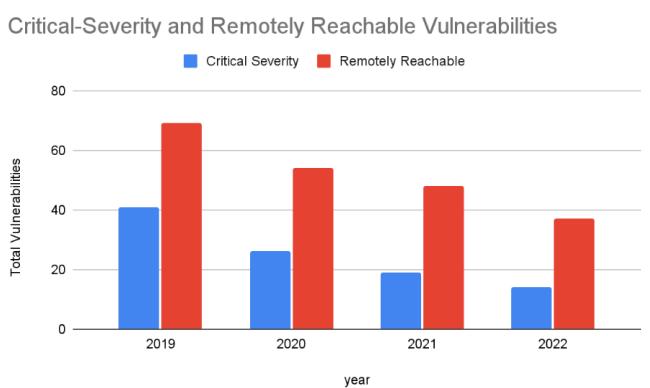
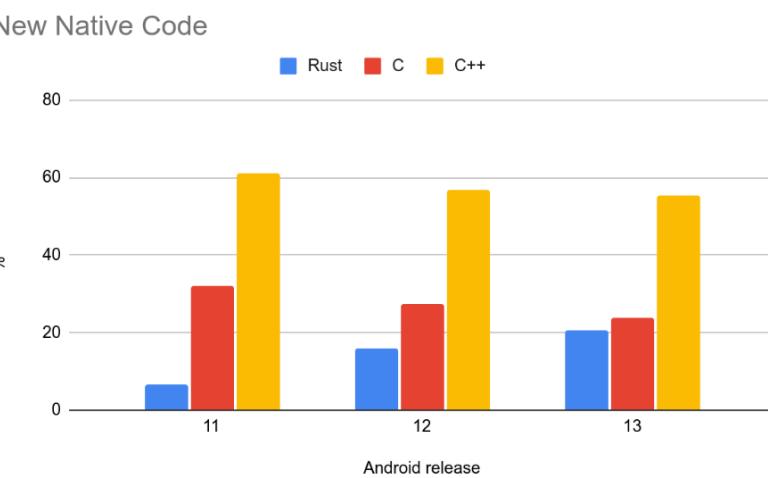
- ▶ Some thread execution timing can cause critical errors, but that is hard to test/detect

CVE	Vendor	Product	Type
CVE-2019-7286	Apple	iOS	Memory Corruption
CVE-2019-7287	Apple	iOS	Memory Corruption
CVE-2019-0676	Microsoft	Internet Explorer	Information Leak
CVE-2019-5786	Google	Chrome	Memory Corruption
CVE-2019-0808	Microsoft	Windows	Memory Corruption
CVE-2019-0797	Microsoft	Windows	Memory Corruption

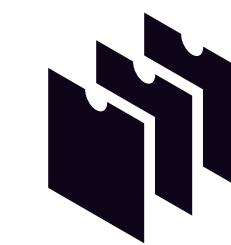
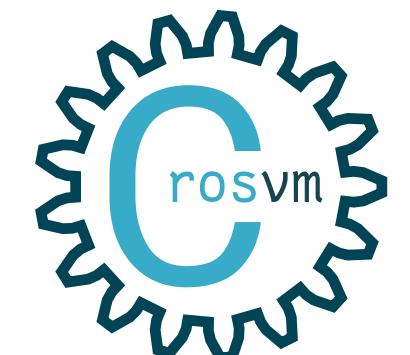


Zero-day attacks in the wild

Rust in Real-World Software

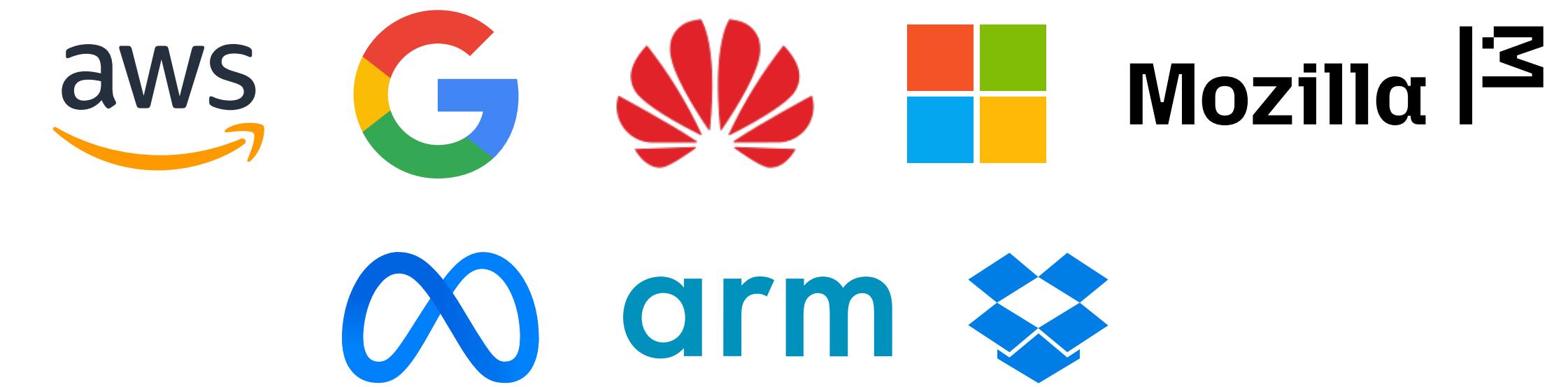


Mozilla |



Rust Foundation

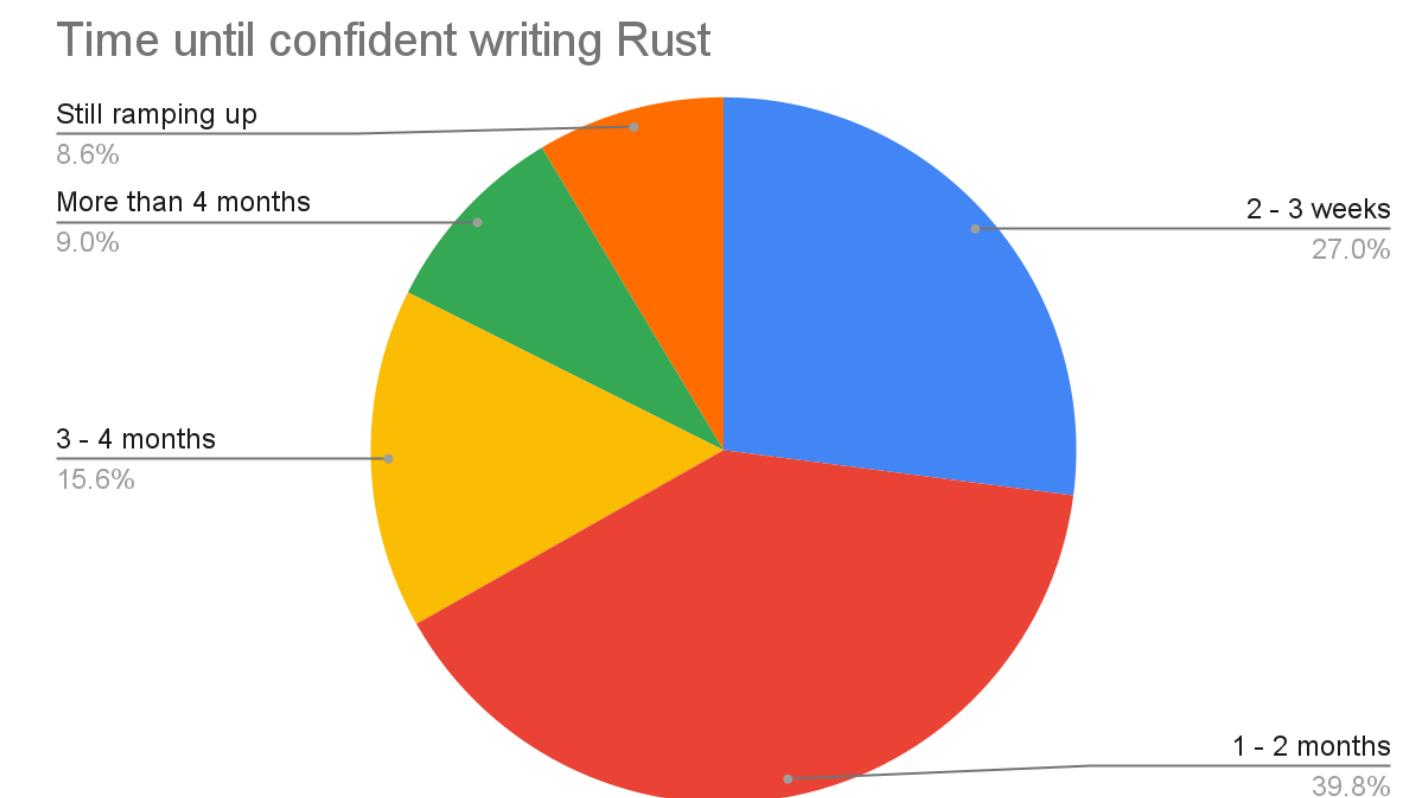
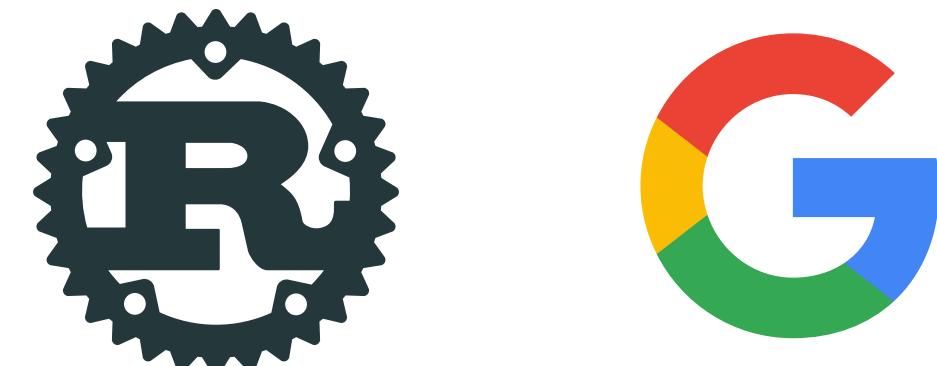
- ♦ NPO for stewarding Rust & its ecosystem
 - ▶ Founded in 2021 by AWS, Google, Huawei, Microsoft & Mozilla, and sponsored by Meta, Arm, Dropbox, JetBrains, ...
 - ▶ Actively donated by tech companies



**Google Contributes \$1M to
Rust Foundation to Support
C++/Rust “Interop Initiative”**

Rust for Google Software Engineers

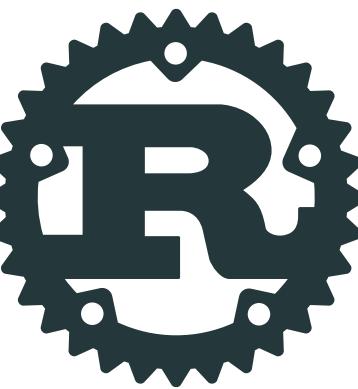
- ♦ A survey on >1k Google software engineers in 2022
 - ▶ ~67% got confident in writing Rust just in 2 months
 - ▶ Over 50% felt as productive in Rust as in other languages in 4 months
 - ▶ ~85% felt more confident in correctness in Rust than in other languages
 - ▶ The top challenges in learning Rust were macros, ownership/borrowing & async



Rust's Community Is Vibrant

◆ Rust has energetic contributors

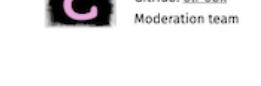
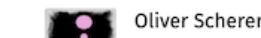
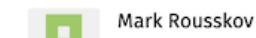
- Language design, semantics, documentation, ...



Leadership council

Charged with the success of the Rust Project as whole, consisting of representatives from top-level teams

Members



Language team

Designing and helping to implement new language features

Members



Operational Semantics team

Working on deciding and specifying the semantics of Rust around unsafe code

Members



Rust's core contributors



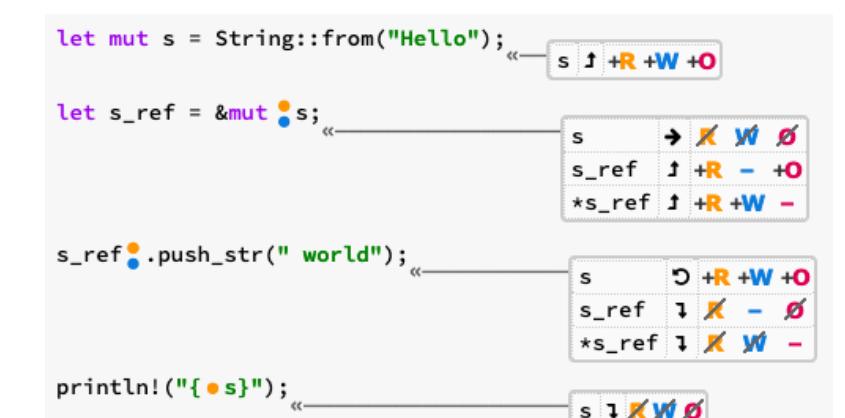
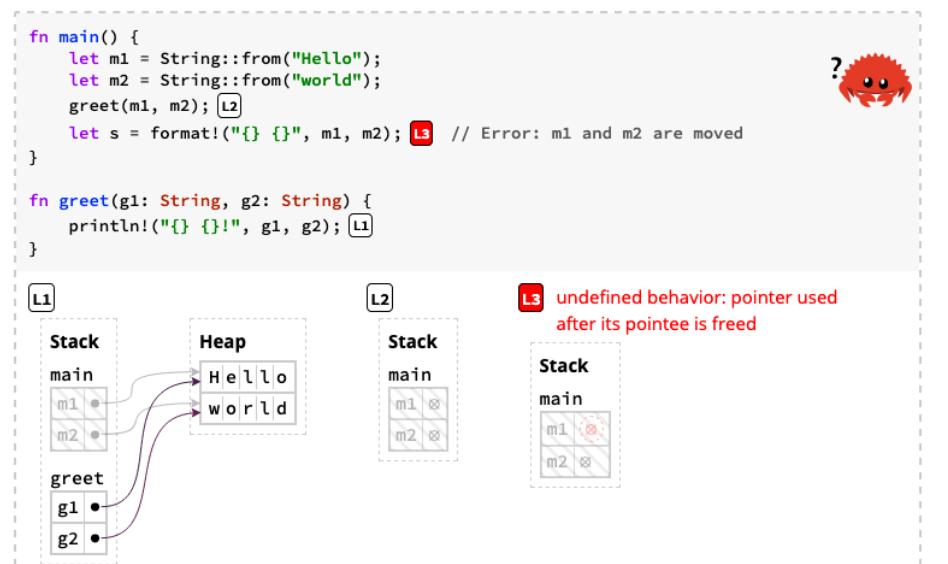
RustFest

A screenshot of a GitHub issue thread titled "Resolve await syntax #57640". The thread shows several comments from users like mzji, chpio, and novacrazy. The comments discuss the implementation of the await macro, comparing it to postfix macros and common lang features. The interface includes standard GitHub UI elements like user avatars, timestamps, and like counts.

Open discussions

References and Borrowing

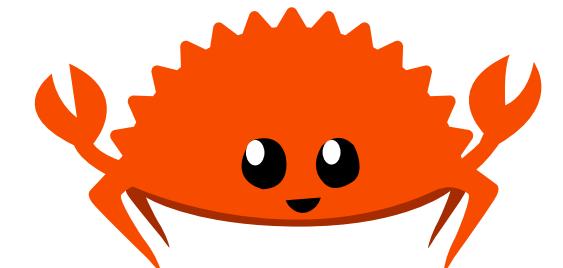
Ownership, boxes, and moves provide a foundation for safely programming with the heap. However, move-only APIs can be inconvenient to use. For example, say you want to read some strings twice:



Documentation

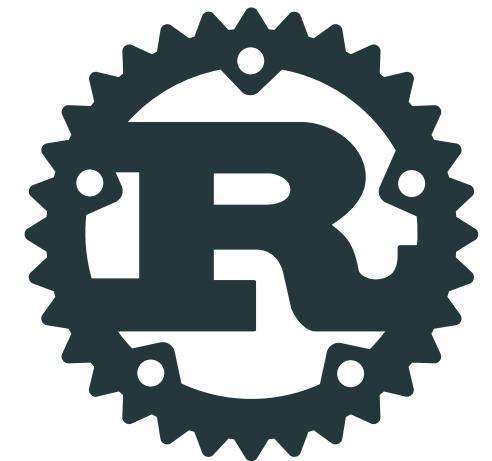
Good Things about Rust

- ◆ **Performance — Comparable to C/C++**
 - ▶ Direct memory & thread operations, no garbage collection
- ◆ **Safety — Unprecedentedly high**
 - ▶ High-level memory & thread safety by innovative **ownership types**
 - The first mainstream language with ownership types
- ◆ **Productivity — By modern features & ecosystem**
 - ▶ Type inferences, traits, hygienic macros, ... / Package managers, LSP, ...



Let's Try Rust Out

♦ Rust Playground <https://play.rust-lang.org/>



The screenshot shows the Rust Playground interface. On the left, the code for a FizzBuzz program is displayed:

```
1 fn main() {
2     for n in 1..=30 {
3         match (n % 3, n % 5) {
4             (0, 0) => println!("FizzBuzz"),
5             (0, _) => println!("Fizz"),
6             (_, 0) => println!("Buzz"),
7             _      => println!("{}" , n),
8         }
9     }
10 }
```

On the right, the execution results are shown in two tabs: 'Execution' and 'Standard Error'. The 'Execution' tab contains the following text:

Compiling playground v0.0.1 (/playground)
Finished `dev` profile [unoptimized + debuginfo]
Running `target/debug/playground`

The 'Standard Output' tab displays the FizzBuzz sequence:

```
1
2
Fizz
4
Buzz
Fizz
7
8
Fizz
Buzz
11
Fizz
```

At the bottom right of the playground interface is a red 'SEND' button.

How to Learn Rust

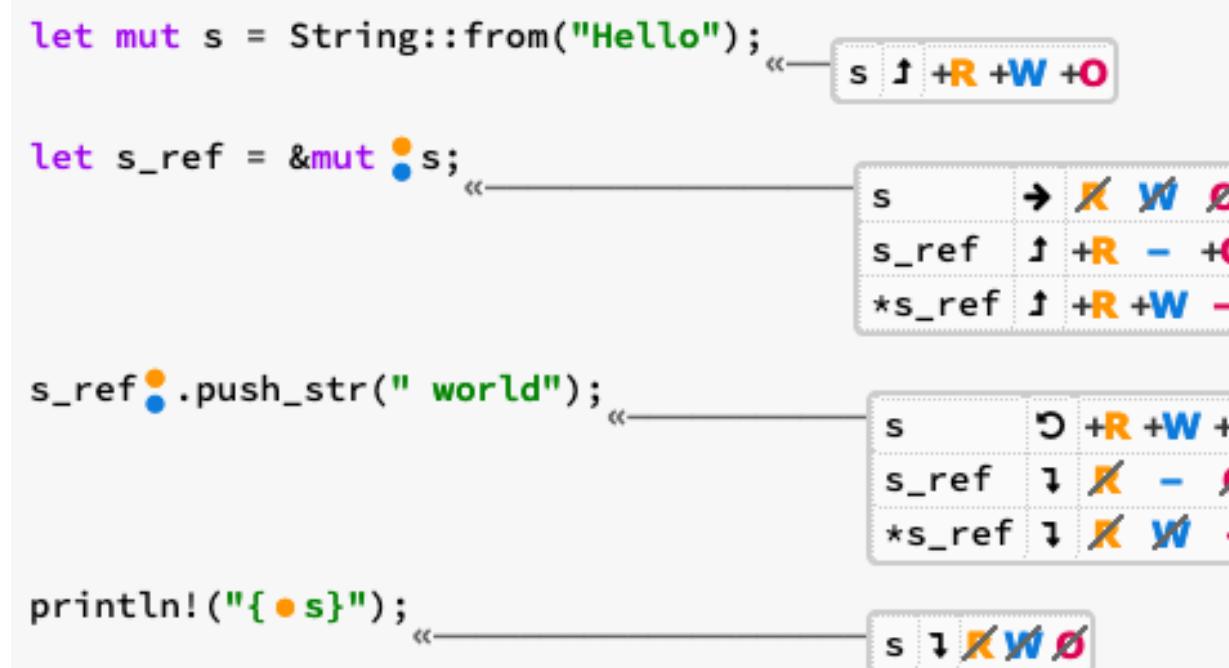
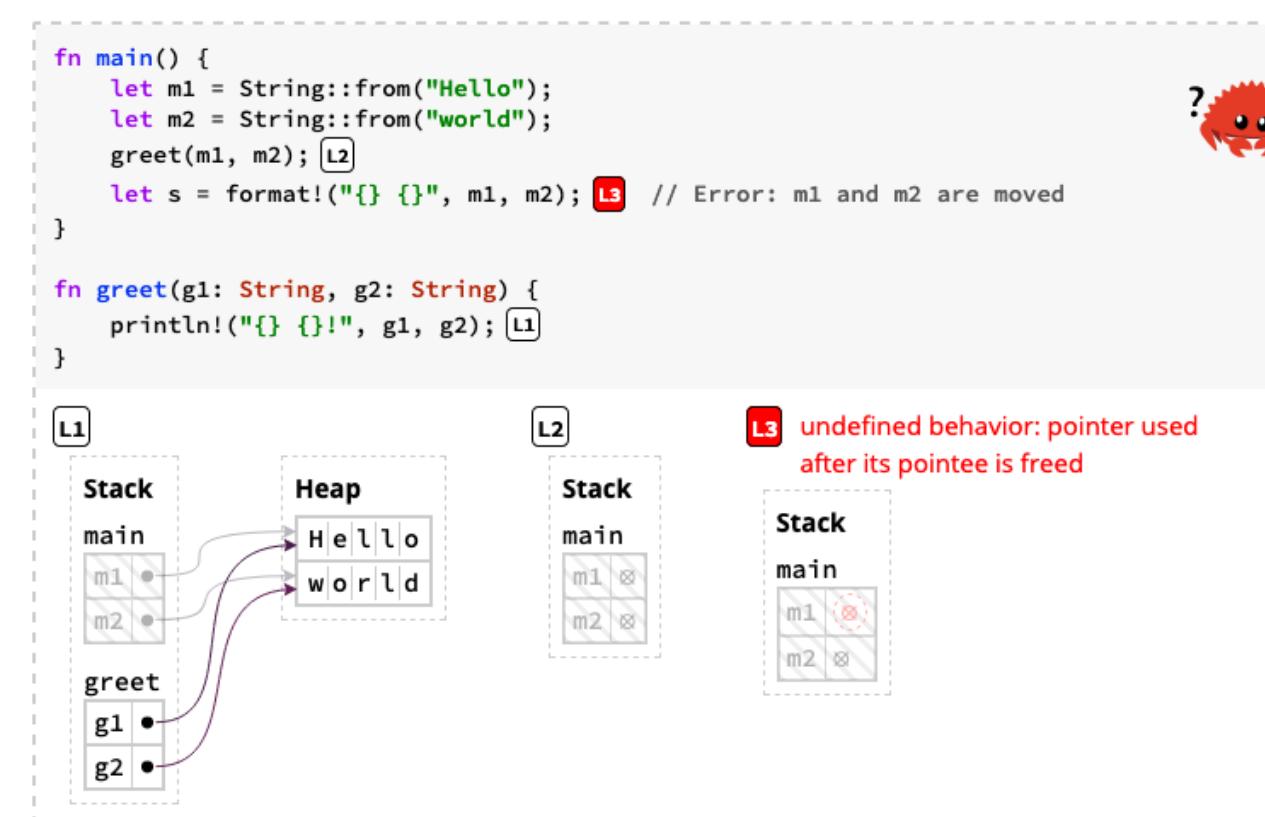
♦ The Rust Book: Standard and comprehensive

- An experimental version with visualization is also available

♦ The Rustonomicon: Digs into Unsafe Rust

References and Borrowing

Ownership, boxes, and moves provide a foundation for safely programming with the heap. However, move-only APIs can be inconvenient to use. For example, say you want to read some strings twice:



Introduction

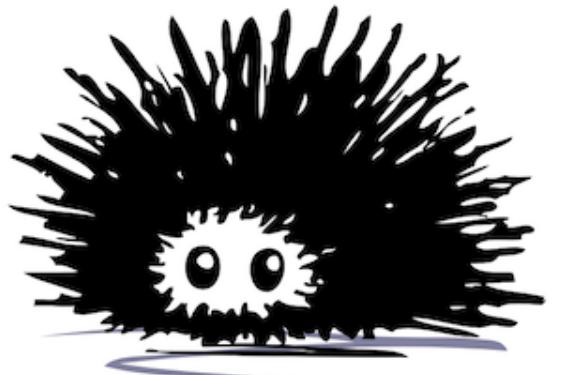
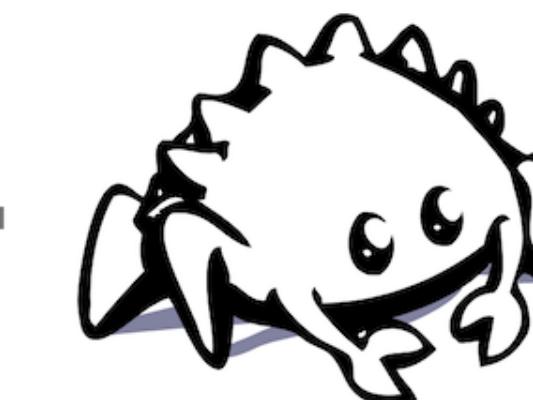
- 1. Meet Safe and Unsafe
 - 1.1. How Safe and Unsafe Interact
 - 1.2. What Unsafe Can Do
 - 1.3. Working with Unsafe
- 2. Data Layout
 - 2.1. repr(Rust)
 - 2.2. Exotically Sized Types
 - 2.3. Other reprs
- 3. Ownership
 - 3.1. References
 - 3.2. Aliasing
 - 3.3. Lifetimes
 - 3.4. Limits of Lifetimes
 - 3.5. Lifetime Elision
 - 3.6. Unbounded Lifetimes
 - 3.7. Higher-Rank Trait Bounds
 - 3.8. Subtyping and Variance
 - 3.9. Drop Check



The Rustonomicon



Meet Safe and Unsafe



It would be great to not have to worry about low-level implementation details. Who could possibly care how much space the empty tuple occupies? Sadly, it sometimes matters and we need to worry about it. The most common reason developers start to care about implementation details is performance, but more importantly, these details can become a matter of correctness when interfacing directly with hardware, operating systems, or other languages.

An experimental version of the Rust Book

The Rustonomicon

#2

Rust's Ownership Types

Big Theme: Divide Difficulties

- ♦ To overcome difficulties, divide them if possible
 - ▶ Then combine the reasoning about the divided parts
 - ▶ Modularity & Composability → Scalability



René Descartes

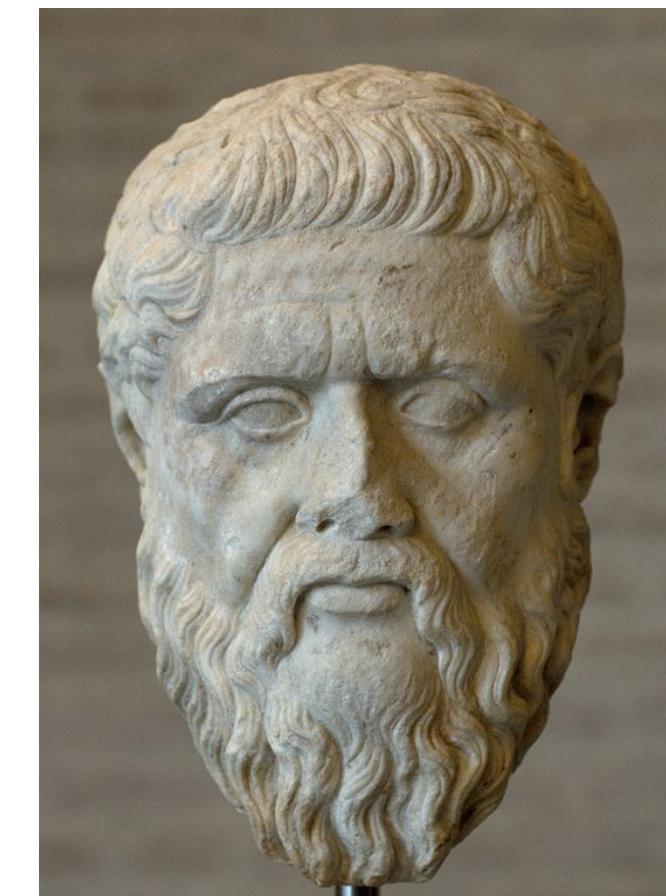
The second, to divide each of the difficulties under examination into as many parts as possible, and as might be necessary for its adequate solution.

Discourse on the Method, Part II

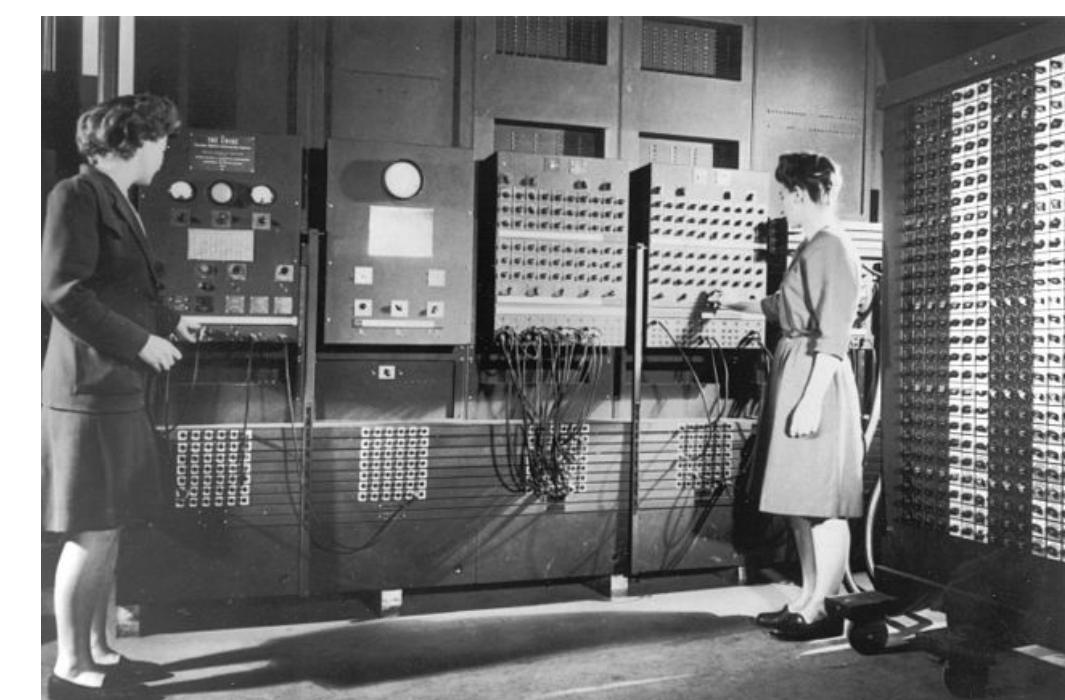
Difficulties in Computation: Mutable State

- ♦ **Ideally, everything is just persistent**
 - ▶ The metaphysical world of the Platonic Ideas
 - ▶ Usual mathematics works in this setting

- ♦ **Physical computation breaks things**
 - ▶ Inevitably deals with mutable state,
e.g., the memory and the environment
 - ▶ Reasoning about mutable state is hard
 - Naive reasoning can break by state update



Plato



Girard's Linear Logic

- ♦ Logic with non-persistent propositions
 - ▶ Can reason about mutable state
 - Applications to computer science



Jean-Yves Girard

LINEAR LOGIC*

“La secrète noirceur du lait . . .”
(Jacques Audiberti)

* Because of its length and novelty this paper has not been subjected to the normal process of refereeing. The editor is prepared to share with the author any criticism that eventually will be expressed concerning this work.

Girard «Linear Logic»
 Theoretical Computer Science 1987

Additive rules:

$\vdash \top, A$ (axiom, A arbitrary) (no rule for $\mathbf{0}$),

$$\frac{\vdash A, C \quad \vdash B, C}{\vdash A \& B, C} \&, \quad \frac{\vdash A, C \quad \vdash B, C}{\vdash A \oplus B, C} 1^\oplus \quad \frac{\vdash B, C}{\vdash A \oplus B, C} 2^\oplus.$$

Multiplicative rules:

$\vdash \mathbf{1}$ (axiom),

$$\frac{\vdash A, C \quad \vdash B, D}{\vdash A \otimes B, C, D} \otimes, \quad \frac{\vdash A, B, C}{\vdash A \wp B, C} \wp.$$

$$\frac{\vdash A}{\vdash \perp, A} \perp,$$

$$\frac{}{\vdash \text{int}, \text{int}} \text{int}.$$

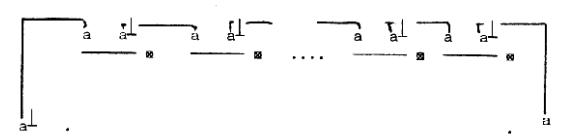


Fig. 58.

Let us explain the interpretation of the proof-net “3”; for this, the best is to return to the *pons asinorum* (Section IV). Specify a coherent space X in order to interpret a in the inner part of the box. Now, $X^1 \wp((!(X \otimes X)) \wp X)$ means $X \rightarrow ((X \rightarrow X) \rightarrow X)$, and should be viewed as the set of all linear maps from X to $!(X \rightarrow X) \rightarrow X$. Then, $!(X \rightarrow X) \rightarrow X$ is $(X \rightarrow X) \rightarrow X$ and should be seen as the set of all stable maps from $X \rightarrow X$ to X . Summing up, what is inside the box describes, for $f = X$, a binary stable function mapping $X \rightarrow X$ into X , and linear in the first argument. This function is defined by $F(x, f) = f(f(x))$, as the semantic computation easily shows. More generally, all proof-nets which represent n are semantically identical: on X , they correspond to the function $F(x, f) = f^n(x)$.

This is very close to the more familiar representation of integers in F ; the main difference is that the argument f is linear. It is possible to stick to ‘ f linear’ because f^n is still linear. But the function associating f^n with f is never linear, but for $n=1$. Now, look at Fig. 59; the diagram represents the successor function, which linearly

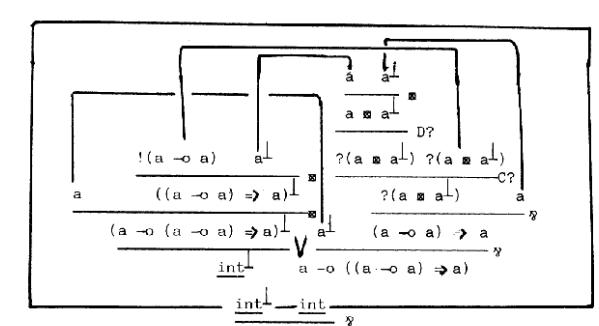


Fig. 59.

maps int into int ; if one applies SUCC to a representation $\#$ of the integer n , by means of the configuration in Fig. 60, then, after normalization, one gets a representation $\# + 1$ of the next integer.

Linear Logic to Rust's Ownership Types

- ♦ Rust's ownership types originate from academic studies
 - ▶ Especially under a strong influence of Cyclone, a safe dialect of C

Linear Logic

Girard 1987



$$\frac{\vdash A, C \quad \vdash B, D}{\vdash A \otimes B, C, D} \otimes, \quad \frac{\vdash A, B, C}{\vdash A \wp B, C} \wp.$$

Linear Types

Wadler 1990

Region Types

Tofte & Talpin 1997

Ownership Types

Clarke+ 1998

Cf. Curry-Howard correspondence
Propositions \Leftrightarrow Types

Cyclone

Grossman+ 2002

Borrows

Rust



2015

The first mainstream language
with ownership types

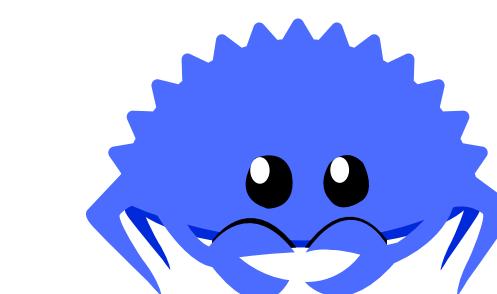
Basic Principle: Aliasing XOR Mutability (AXM)

- ♦ In principle, each region is either aliased xor mutable
 - No data races → Memory & thread safety
- ♦ Ownership = Access permission

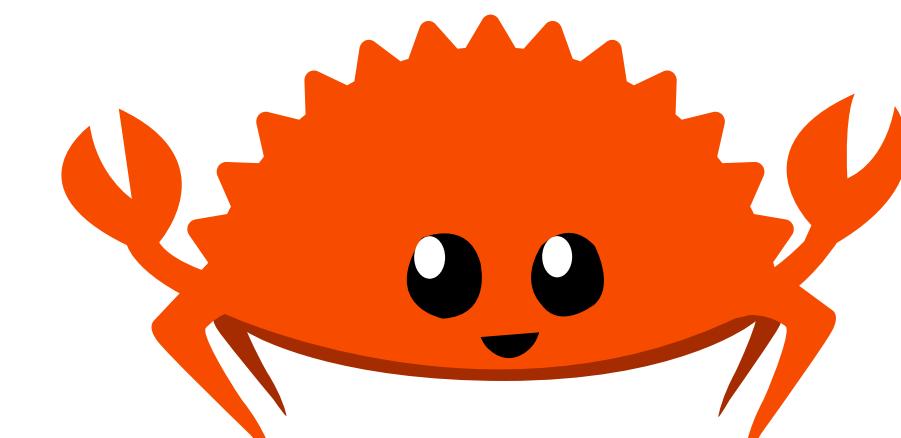
Aliased & Immutable



Read-only



Mutable & Exclusive



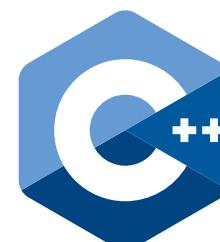
Writable

Unique

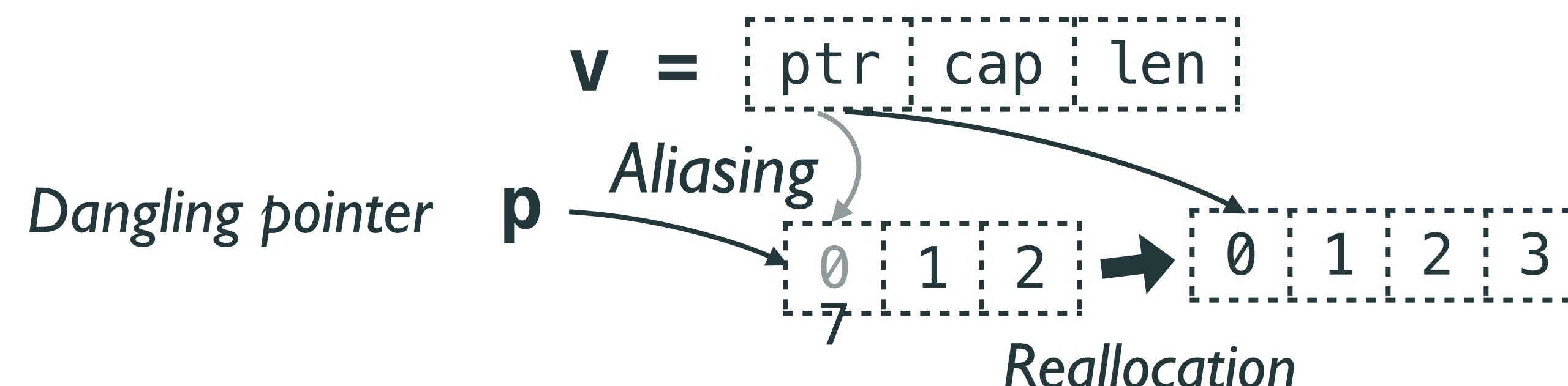
Aliasing AND Mutability Is Dangerous

- ♦ Aliased mutable state is the core source of bugs
 - ▶ It can cause critical data races

Example A dangling pointer caused by a data race



```
vector<int> v { 0, 1, 2 }; int *p = &v[0];  
v.push_back(3); *p = 7; // Data race!  
printf("%d\n", v[0]); // Prints 0, not 7
```



This can even cause
a buffer overflow!

Rust Bans Aliasing AND Mutability

- ♦ Rust's ownership types ban Aliasing AND Mutability
 - No data races → Memory & thread safety

Example



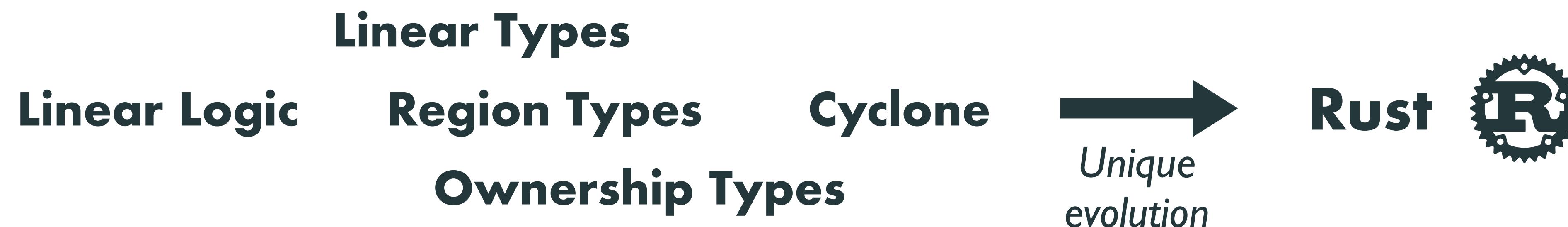
Rejected

```
let mut v = vec![0, 1, 2]; let p = &mut v[0];
v.push(3); *p = 7; // OWNERSHIP ERROR
println!("{}", v[0]);
```

error[E0499]: cannot borrow `v` as mutable more than once at a time
| ...; let p = &mut v[0];
| - first mutable borrow occurs here
| v.push(3); *p = 7; // Data race!
| ^ ----- first borrow later used here
| second mutable borrow occurs here

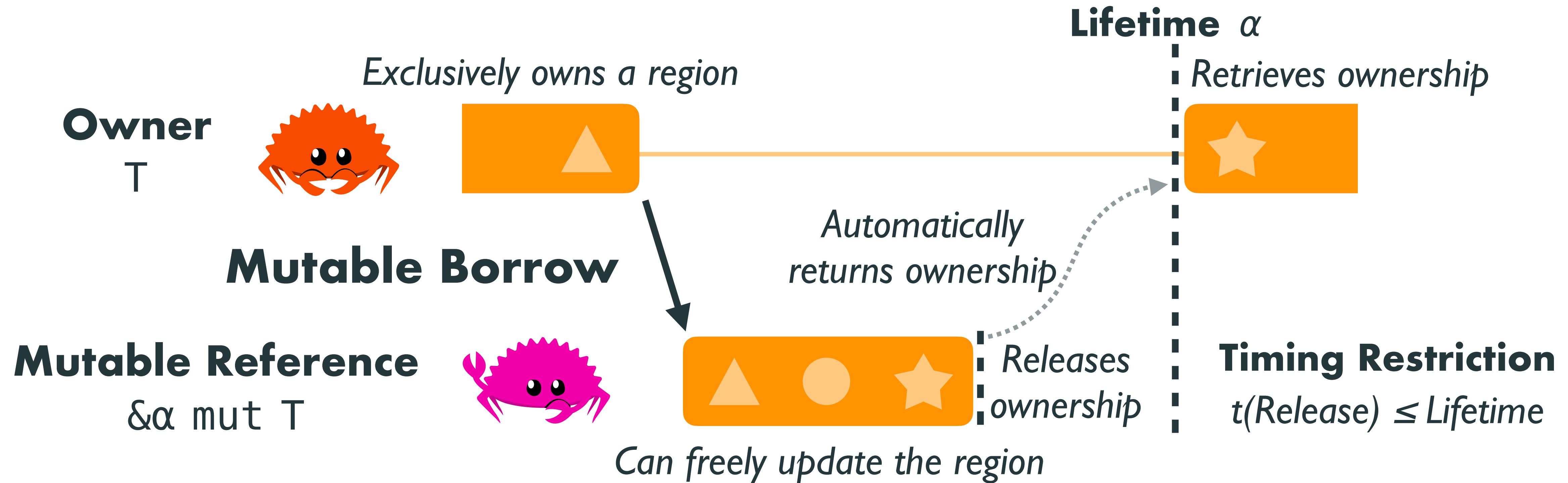
Overview of Rust's Ownership Type System

- ♦ **Statically checks Aliasing XOR Mutability**
 - ▶ Guarantees: No data races → Memory & thread safety
 - ▶ Automated, lightweight, and programmer-friendly
- ♦ **3 keys to success: Borrows, Unsafe, Interior mutability**
 - ▶ What makes Rust different from more traditional approaches



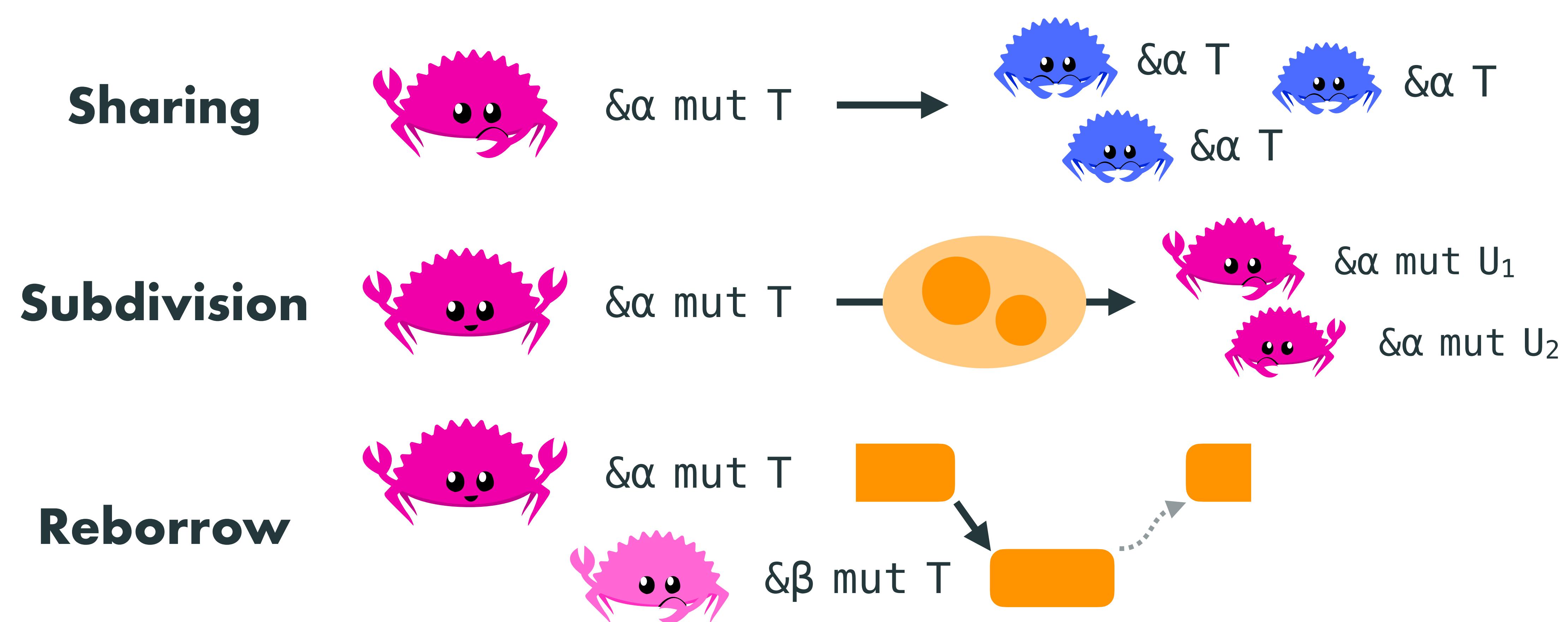
Rust's Key #1: Borrows

- ◆ Temporarily borrow ownership
 - ▶ No direct communications needed when releasing ownership



Operations on Borrows

- ◆ Various high-level operations on borrows are provided

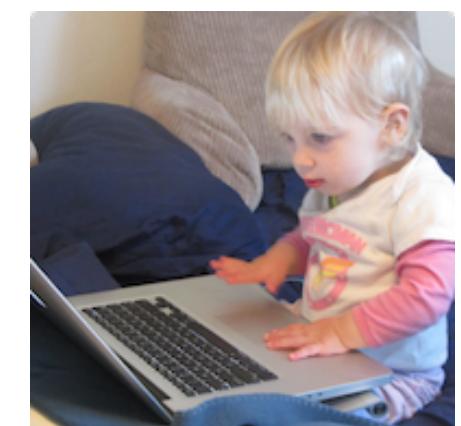


Borrow Checking

- ◆ Automatic static checking on borrows
 - ▶ Esp. the timing restriction $t(\text{Release}) \leq \text{Lifetime}$
- ◆ Actively evolving over time
 - ▶ Older (~2018) — Scope-based, lexical lifetimes
 - $t(\text{Release})$ is the end of the scope
 - ▶ Now — Non-lexical lifetimes by Niko Matsakis
 - $t(\text{Release})$ is inferred by liveness analysis
 - ▶ Future? — “Borrow checker within”
 - More info in types, self-borrows supported



Niko Matsakis



His blog
“baby steps”

```
fn new_widget(&self, name: String) -> Widget {  
    let name_suffix: &'name str = &name[3..];  
        // --- borrowed from "name"  
    let model_prefix: &'self.model str = &self.model[..2];  
        // ----- borrowed from "self.model"  
}
```

Rust's Key #2: Unsafe

- ♦ **Unsafe** = Back door against static checking
 - ▶ In particular, operations on raw pointers (`*mut T`, `*const T`) that are not protected by static ownership checks
 - ▶ Key pattern: Encapsulate **unsafe** implementations into **safe APIs**



Example

```
struct Vec<T> {  
    ptr : *mut T,  
    len : uint,  
    cap : uint  
}
```



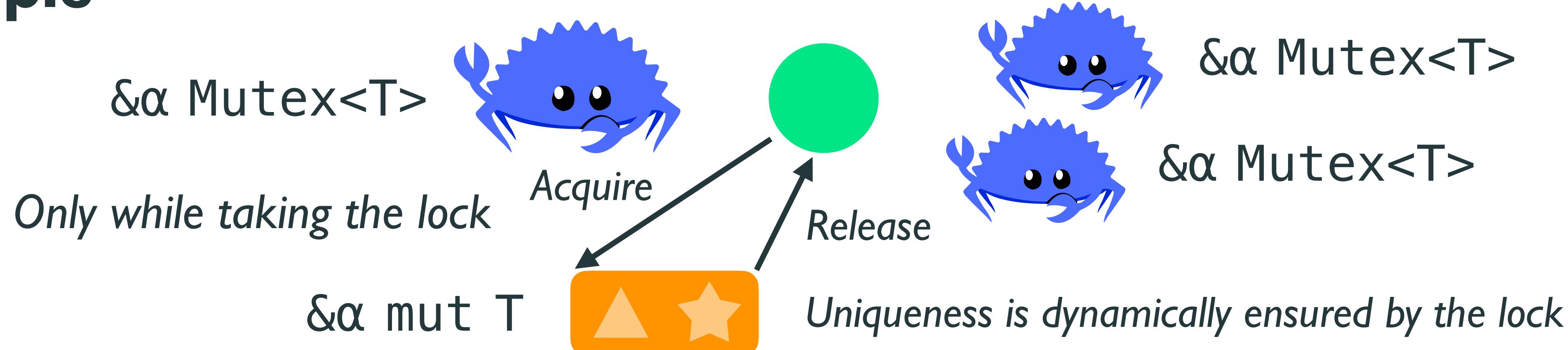
```
Safe API  
fn index_mut<&mut T>(v : &α mut Vec<T>,  
                      i : uint) -> &α mut T {  
    assert!(i < v.len);  
    unsafe { v.ptr.offset(i) }  
}
```

Raw pointer manipulation

Rust's Key #3: Interior Mutability

- ♦ **Mutability conditionally allowed for shared borrows**
 - ▶ Usual types (int, String, ...): Just immutable when shared
 - ▶ Special types (Mutex, RwLock, Arc, Cell, ...) have interior mutability
 - Through carefully designed APIs, to satisfy the AXM principle

Example



#3

Formal Verification for Rust

Recap: Formal Verification

♦ Proving that the software satisfies the specs

- ▶ A rigorous way to eradicate software bugs
- ▶ Practically used in critical domains
 - Aerospace, finance, security, etc.

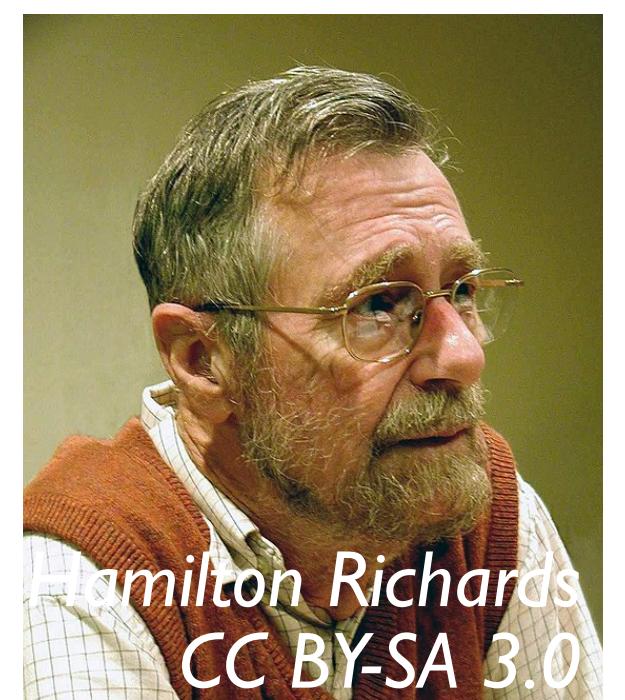


♦ A central topic in software science

- ▶ Program logics, especially Hoare Logic (1969)
 - Dijkstra's weakest precondition calculus (1975)
- ▶ Advanced types can also be used



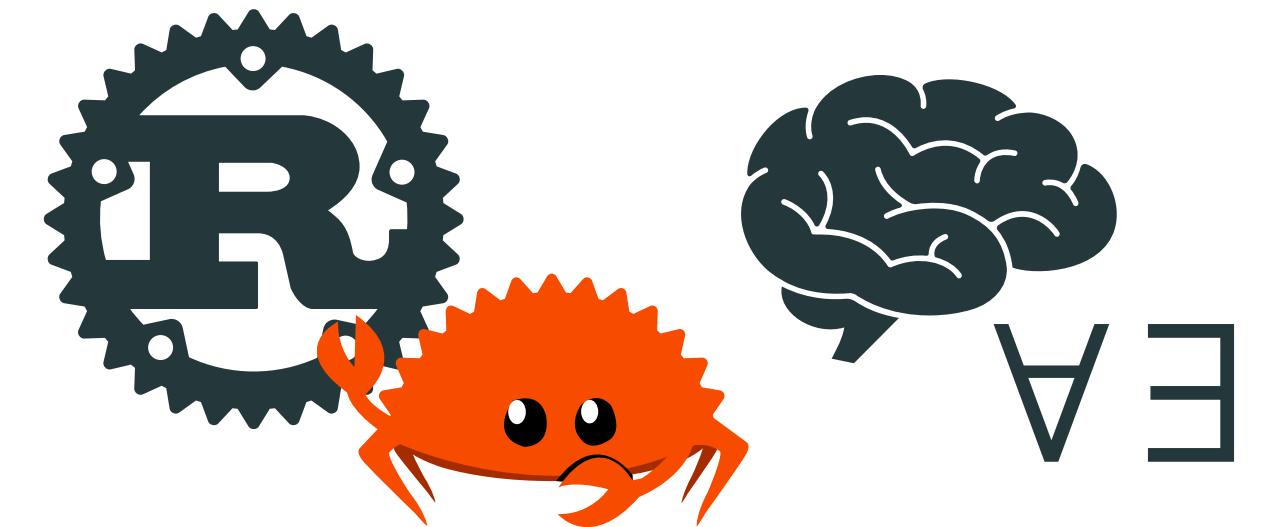
Tony Hoare



Edsger Dijkstra

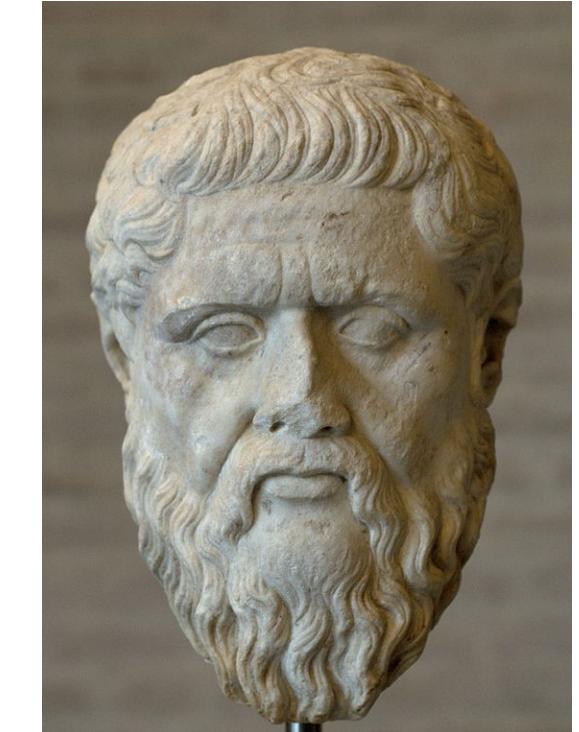
Goal: Formal Verification of Rust Programs

- ◆ **Want to verify Rust programs formally**
 - ▶ Significant in the real world
 - Rust is often used for foundational software, which various applications are based on
 - Operating systems, servers, crypto libraries, numerical libraries, solvers, ...
 - ▶ Esp. the functional correctness
 - The correctness of the output value over every input value
 - ▶ We can take advantage of Rust's ownership types
 - Rust's ownership types exclude bad situations



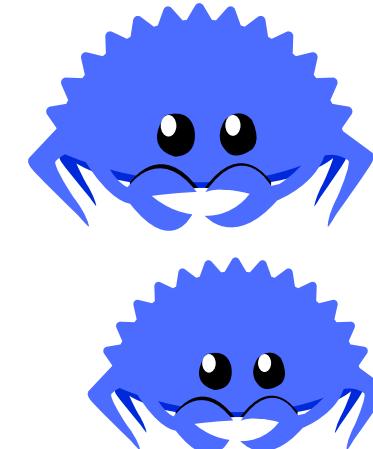
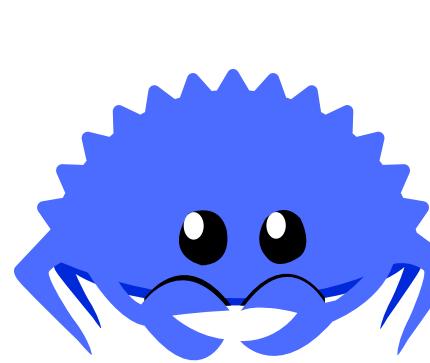
Basic Idea – Turn into Functional Models

- ◆ Turn Rust programs into functional models, leveraging Rust's ownership types
 - ▶ Into models without the global memory state
 - Into the “metaphysical” world where everything is persistent



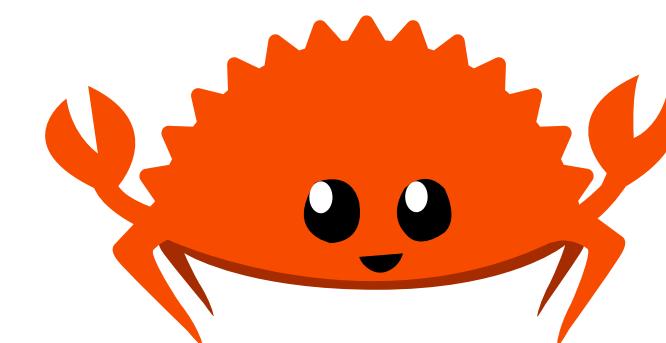
Plato

Aliased & Immutable



Just an immutable value

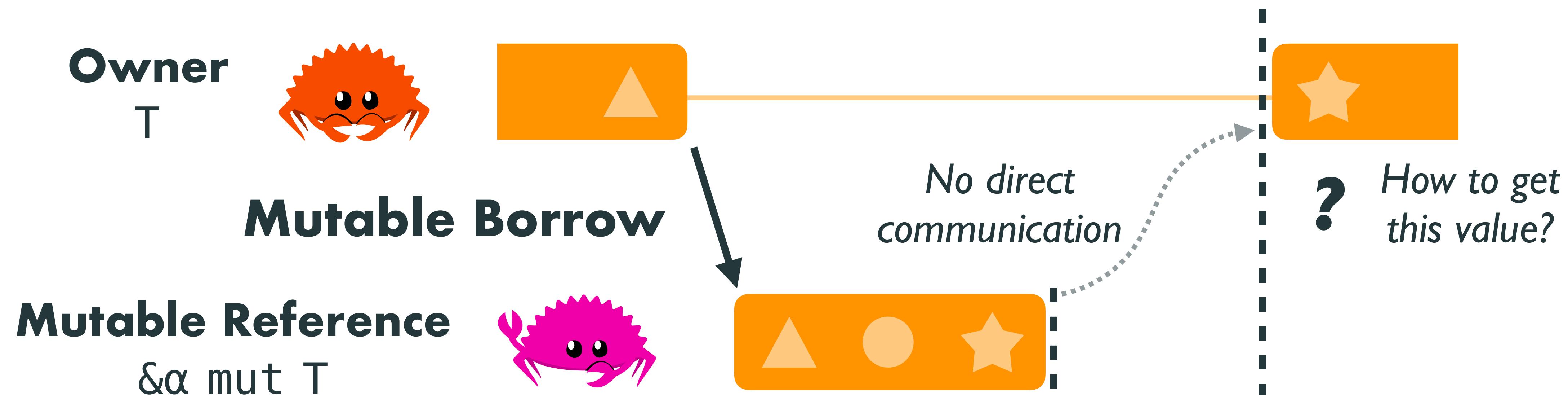
Fully Own



Can track the value change

Challenge: Mutable Borrows

- ♦ **Mutable borrows are hard for functional reasoning**
 - ▶ How to “tell” the result of the borrower’s mutation to the lender?
 - There is no direct communication when ownership is returned



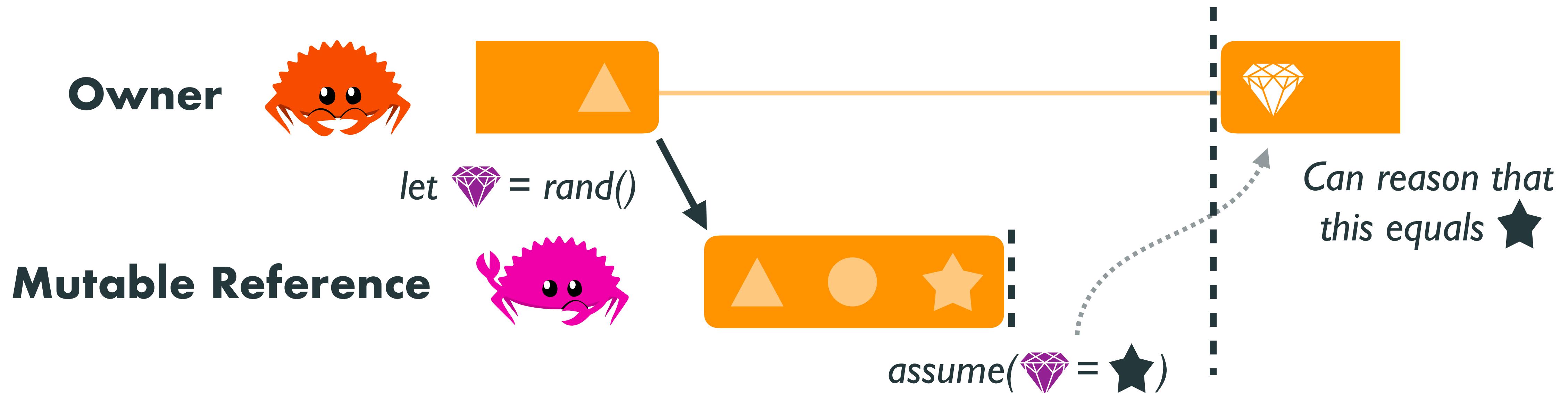
Solution: My Work, RustHorn



[Matsushita+
ESOP 2020 / TOPLAS 2021]

♦ Model mutable borrows by prophecy 💎

- ▶ Prophecy [Abadi & Lamport 1988]: What fetches info about the future
 - A kind of time-machine reasoning!
- ▶ Specifically: Prophecy of the result of the borrower's mutation



Power of RustHorn's Approach

- ◆ **Prophecy** can naturally model borrow operations

- ▶ By resolving prophecies when releasing ownership
 - For advanced operations, prophecies are resolved partially



Example fn push<α, T>(v : &α mut Vec<T>, a : T)

ensures $\wedge v = *v ++ a$ *Resolve the prophecy*

fn index_mut<α, T>(v : &α mut Vec<T>, i : uint) -> &α mut T

requires i < v.len()

ensures *res = v[i] $\wedge v = *v \{ i := \wedge res \}$

Partially resolve the prophecy

RustHorn & Its Descendants

- ◆ **RustHorn [M+ 2020/21] — Fully automated Rust verifier**
 - ▶ By turning Rust code into logic formulas with fixed points
 - Interesting examples around recursive data types [V.K.+ POPL 2022]
- ◆ **Creusot [Denis+ 2022] — Semi-automated Rust verifier**
 - ▶ Used for verifying a SAT solver written in Rust [Skotåm 2022]!
- ◆ **RusSOL [Fiala+ PLDI 2023] — Rust program synthesizer**
 - ▶ Synthesizes a Rust program that satisfies RustHorn-style specs



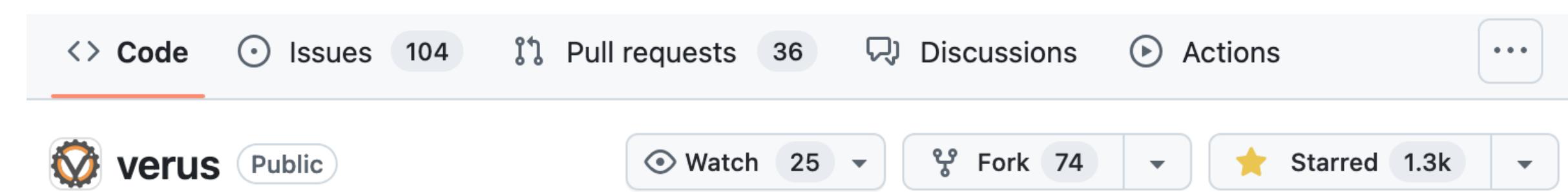
Foundational Verification of Rust

- ♦ **Foundational verification of Rust is also studied** 
- ▶ Rust itself is non-trivial and thus should be verified
 - Including Rust APIs encapsulating unsafe code
- ▶ RustBelt [Jung+ POPL 2018] — Foundation for Rust 
 - Proved memory & thread safety guarantees of Rust's ownership types
 - Modeled Rust's ownership types in Iris Separation Logic [Jung+ POPL 2015]
- ▶ RustHornBelt [Matsushita+ PLDI 2022] — Foundation for RustHorn 
 - Extended RustBelt to prove RustHorn-style reasoning sound

New Rust Verifier, Verus

[Lattuada+ OOPSLA 2023 / SOSP 2024]

- ◆ Powerful semi-automated Rust program verifier
 - ▶ From Microsoft Research, very actively developed
 - Playground, LSP, standard libraries, tutorials & references, ...
 - ▶ Extends Rust's ownership type system with an SMT solver
 - Can flexibly verify unsafe Rust code in a style like separation logic
 - ▶ Already applied to practical Rust verification projects
 - 2 (out of 3) best papers of OSDI 2024 used Verus



```
use builtin::*;

fn m(en)
    m <= x,
    m <= y,
    m == x || m == y,
{
    if x <= y {
        y
    } else {
        y
    }
}

} // verus!
```

Let's Try Verus Out

- ♦ Verus Playground <https://play.verus-lang.org/>
- Just write code with assertions and push “Verify”!



The image shows the Verus Playground interface with two code editor panes and their execution results.

Left Editor:

```
1 use vstd::prelude::*;

2 verus! {

3     spec fn min(x: int, y: int) -> int {
4         if x <= y { x } else { y }
5     }

6     fn test() {
7         assert(forall |i|
8             min(i, i) == i);
9         assert(forall |i, j|
10            min(i, j) <= i &&
11            min(i, j) <= j);
12    }
13 }
14 }
```

Execution Results:

Execution tab: Standard Error
Standard Output
verification results:: 1 verified, 0 errors

Right Editor:

```
1 use vstd::prelude::*;

2 verus! {

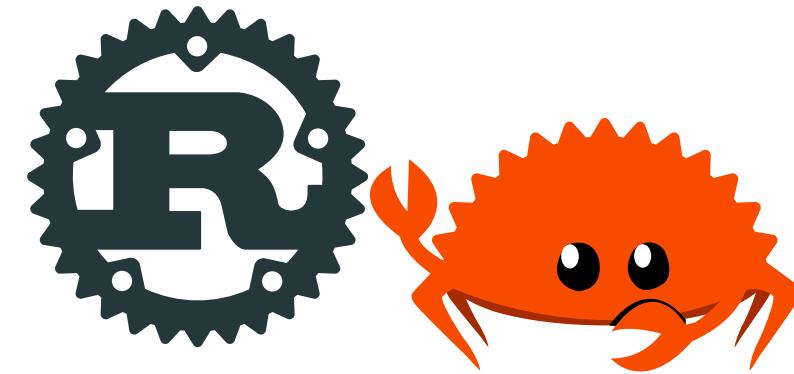
3     spec fn min(x: int, y: int) -> int {
4         if x <= y { x } else { y }
5     }

6     fn test() {
7         assert(forall |i|
8             min(i, i) == i);
9         assert(forall |i, j|
10            min(i, j) <= i &&
11            min(i, j) <= j);
12    }
13 }
14 }
```

Execution Results:

error: assertion failed
--> /playground/src/main.rs:12:12
| assert(forall |i, j|
|-----^
| min(i, j) <= i &&
| min(i, j) < j;
|-----^ assertion failed
note: diagnostics via expansion:
forall |i, j|
min(i, j) <= i ✓
min(i, j) < j ✗
--> /playground/src/main.rs:14:9
| min(i, j) < j;
| ^^^^^^

Hopes for the Future

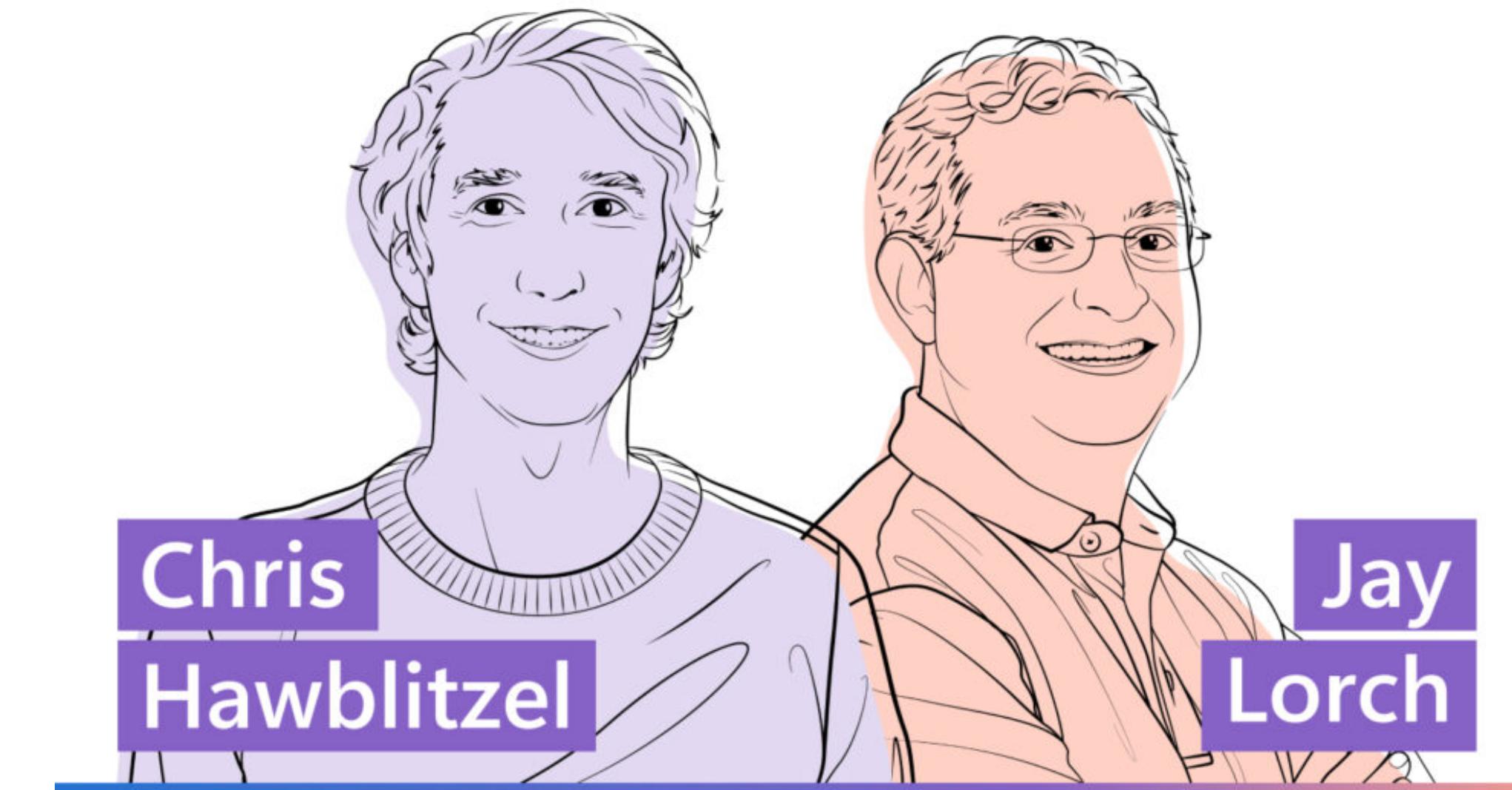


Practical, High-Performance Verification in Rust

[Overview](#) [People](#) [Publications](#) [Groups](#) [Microsoft Research blog](#)

Formal verification is a promising approach to eliminate bugs at compile time, before software ships. Unfortunately, verifying the correctness of system software traditionally requires heroic developer effort. In this project, we aim to enable accessible, faster, cheaper verification of rich properties for realistic systems written in Rust using Verus.

Verus is an SMT-based tool for formally verifying Rust programs. With Verus, programmers express proofs and specifications using the Rust language, with no need to learn a new language. At the same time, Verus takes advantage of Rust's linear types and borrow checking to express ownership and separation in proofs. We are using Verus to develop high-performance, verifiably correct systems. We are also exploring the use of Large Language Models to further ease the effort of developing proof with Verus.



Verus: A Practical Foundation for Systems Verification

Andrea Lattuada*
MPI-SWS

Matthias Brun
ETH Zurich

Pranav Srinivasan
University of Michigan

Chris Hawblitzel
Microsoft Research

Oded Paden*
Weizmann Institute of Science

Travis Hance
Carnegie Mellon University

Chanhee Cho
Carnegie Mellon University

Reto Achermann
University of British Columbia

Jon Howell
VMware Research

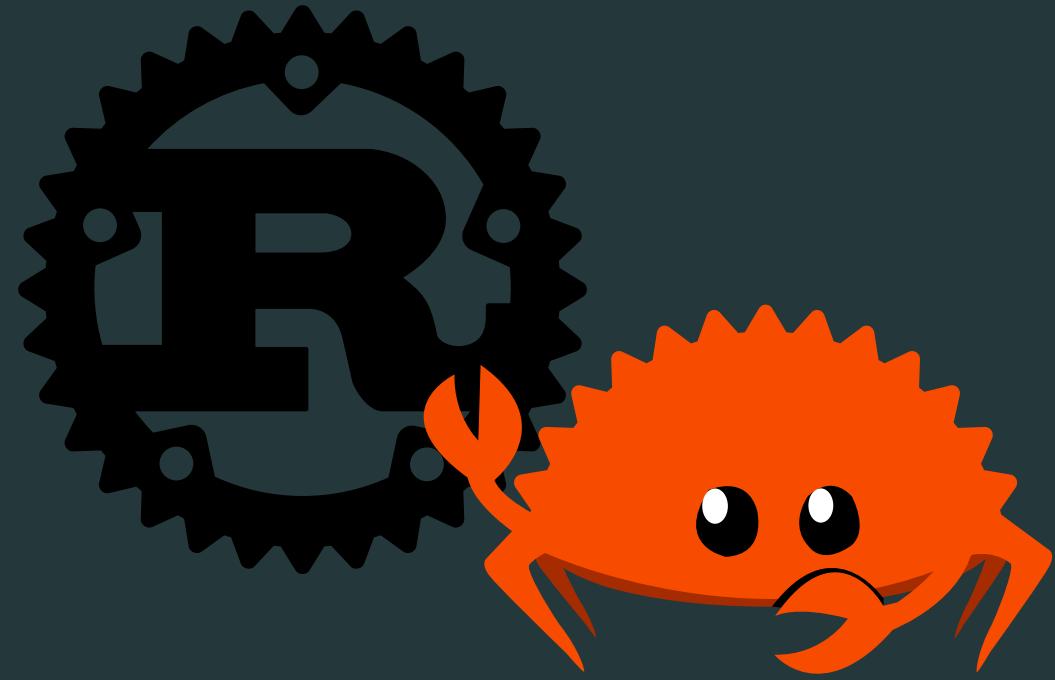
Bryan Parno
Carnegie Mellon University

Jay Bosamiya†
Microsoft Research

Hayley LeBlanc
University of Texas at Austin

Tej Chajed
University of Wisconsin-Madison

Jacob R. Lorch
Microsoft Research



Rust Takes Us to the New Age of Software Development

