RustBelt: Securing the Foundations of the Rust Programming Language

● *Adam McDaniel* ● *Alexander Krneta* ● *Matthew Jones* ● *Kapildev Neupane* ●

Brief Overview: Rust's Type System Formally Verified

- **The selected paper for this presentation covers two main topics:**

    1. The history of adding safety to C-like languages. Rust was born out of research exploring a safe dialect of C: Cyclone.

    2. The paper presents a formal proof for Rust's type system: Rust's memory safety guarantees are 100% sound, checked using the **Coq** theorem proving assistant.

- Ultimately, this paper is about demonstrating the real developer productivity benefits of Rust's memory model.

Rules for lifetimes: $(\Gamma \mid \mathbf{E};\mathbf{L} \vdash \kappa_1 \sqsubseteq \kappa_2$ and $\Gamma \mid \mathbf{E};\mathbf{L} \vdash \kappa$ alive$)$

Lincl-static
$$\Gamma \mid \mathbf{E};\mathbf{L} \vdash \kappa \sqsubseteq \mathbf{static}$$

Lincl-local
$$\frac{\kappa \sqsubseteq_l \overline{\kappa} \in \mathbf{L} \qquad \kappa' \in \overline{\kappa}}{\Gamma \mid \mathbf{E};\mathbf{L} \vdash \kappa \sqsubseteq \kappa'}$$

Lincl-extern
$$\frac{\kappa \sqsubseteq_e \kappa' \in \mathbf{E}}{\Gamma \mid \mathbf{E};\mathbf{L} \vdash \kappa \sqsubseteq \kappa'}$$

Lincl-refl
$$\Gamma \mid \mathbf{E};\mathbf{L} \vdash \kappa \sqsubseteq \kappa$$

Lincl-trans
$$\frac{\Gamma \mid \mathbf{E};\mathbf{L} \vdash \kappa \sqsubseteq \kappa' \qquad \Gamma \mid \mathbf{E};\mathbf{L} \vdash \kappa' \sqsubseteq \kappa''}{\Gamma \mid \mathbf{E};\mathbf{L} \vdash \kappa \sqsubseteq \kappa''}$$

Lalive-local
$$\frac{\kappa \sqsubseteq_l \overline{\kappa} \in \mathbf{L} \qquad \forall i.\ \mathbf{E};\mathbf{L} \vdash \overline{\kappa}_i \text{ alive}}{\Gamma \mid \mathbf{E};\mathbf{L} \vdash \kappa \text{ alive}}$$

Lalive-incl
$$\frac{\Gamma \mid \mathbf{E};\mathbf{L} \vdash \kappa \text{ alive} \qquad \Gamma \mid \mathbf{E};\mathbf{L} \vdash \kappa \sqsubseteq \kappa'}{\mathbf{E};\mathbf{L} \vdash \kappa' \text{ alive}}$$

Rules for subtyping and type coercions: $(\Gamma \mid \mathbf{E};\mathbf{L} \vdash \tau_1 \Rightarrow \tau_2$ and $\Gamma \mid \mathbf{E};\mathbf{L} \vdash \mathbf{T}_1 \overset{\text{ctx}}{\Rrightarrow} \mathbf{T}_2)$

T-bor-lft
$$\frac{\Gamma \mid \mathbf{E};\mathbf{L} \vdash \kappa \sqsubseteq \kappa'}{\Gamma \mid \mathbf{E};\mathbf{L} \vdash \&_\mu^{\kappa'} \tau \Rightarrow \&_\mu^{\kappa} \tau}$$

C-subtype
$$\frac{\Gamma \mid \mathbf{E};\mathbf{L} \vdash \tau \Rightarrow \tau'}{\Gamma \mid \mathbf{E};\mathbf{L} \vdash p \lhd \tau \overset{\text{ctx}}{\Rrightarrow} p \lhd \tau'}$$

C-copy
$$\frac{\tau \text{ copy}}{\Gamma \mid \mathbf{E};\mathbf{L} \vdash p \lhd \tau \overset{\text{ctx}}{\Rrightarrow} p \lhd \tau, p \lhd \tau}$$

C-split-own
$$\mathbf{E};\mathbf{L} \vdash p \lhd \mathbf{own}_n\, \tau_1 \times \tau_2 \overset{\text{ctx}}{\Leftrightarrow} p.0 \lhd \mathbf{own}_n\, \tau_1, \lhd \mathbf{own}_n\, \tau_2$$

C-share
$$\frac{\Gamma \mid \mathbf{E};\mathbf{L} \vdash \kappa \text{ alive}}{\Gamma \mid \mathbf{E};\mathbf{L} \vdash p \lhd \&_{\mathbf{mut}}^{\kappa} \tau \overset{\text{ctx}}{\Rrightarrow} p \lhd \&_{\mathbf{shr}}^{\kappa} \tau}$$

C-borrow
$$\Gamma \mid \mathbf{E};\mathbf{L} \vdash p \lhd \mathbf{own}_n\, \tau \overset{\text{ctx}}{\Rrightarrow} p \lhd \&_{\mathbf{mut}}^{\kappa} \tau, p \lhd^{\dagger\kappa} \mathbf{own}_n\, \tau$$

C-reborrow
$$\frac{\Gamma \mid \mathbf{E};\mathbf{L} \vdash \kappa' \sqsubseteq \kappa}{\Gamma \mid \mathbf{E};\mathbf{L} \vdash p \lhd \&_{\mathbf{mut}}^{\kappa} \tau \overset{\text{ctx}}{\Rrightarrow} p \lhd \&_{\mathbf{mut}}^{\kappa'} \tau, p \lhd^{\dagger\kappa'} \&_{\mathbf{mut}}^{\kappa} \tau}$$

Rules for reading and writing: $(\Gamma \mid \mathbf{E};\mathbf{L} \vdash \tau_1 \circ\!\!-^{\tau} \tau_2$ and $\Gamma \mid \mathbf{E};\mathbf{L} \vdash \tau_1 -\!\!\circ^{\tau} \tau_2)$

Tread-own-copy
$$\frac{\tau \text{ copy}}{\Gamma \mid \mathbf{E};\mathbf{L} \vdash \mathbf{own}_n\, \tau \circ\!\!-^{\tau} \mathbf{own}_n\, \tau}$$

Tread-own-move
$$\frac{n = \text{size}(\tau)}{\Gamma \mid \mathbf{E};\mathbf{L} \vdash \mathbf{own}_m\, \tau \circ\!\!-^{\tau} \mathbf{own}_m\, \lightning n}$$

Tread-bor
$$\frac{\tau \text{ copy} \qquad \Gamma \mid \mathbf{E};\mathbf{L} \vdash \kappa \text{ alive}}{\Gamma \mid \mathbf{E};\mathbf{L} \vdash \&_\mu^{\kappa} \tau \circ\!\!-^{\tau} \&_\mu^{\kappa} \tau}$$

Twrite-own
$$\frac{\text{size}(\tau) = \text{size}(\tau')}{\Gamma \mid \mathbf{E};\mathbf{L} \vdash \mathbf{own}_n\, \tau' -\!\!\circ^{\tau} \mathbf{own}_n\, \tau}$$

Twrite-bor
$$\frac{\Gamma \mid \mathbf{E};\mathbf{L} \vdash \kappa \text{ alive}}{\Gamma \mid \mathbf{E};\mathbf{L} \vdash \&_{\mathbf{mut}}^{\kappa} \tau -\!\!\circ^{\tau} \&_{\mathbf{mut}}^{\kappa} \tau}$$

Rules for typing of instructions: $(\Gamma \mid \mathbf{E};\mathbf{L} \mid \mathbf{T} \vdash I \dashv x.\, \mathbf{T}_2)$

S-num
$$\Gamma \mid \mathbf{E};\mathbf{L} \mid \emptyset \vdash z \dashv x.\, x \lhd \mathbf{int}$$

S-nat-leq
$$\Gamma \mid \mathbf{E};\mathbf{L} \mid p_1 \lhd \mathbf{int}, p_2 \lhd \mathbf{int} \vdash p_1 \leq p_2 \dashv x.\, x \lhd \mathbf{bool}$$

S-new
$$\Gamma \mid \mathbf{E};\mathbf{L} \mid \emptyset \vdash \mathtt{new}(n) \dashv x.\, x \lhd \mathbf{own}_n\, \lightning n$$

S-delete
$$\frac{n = \text{size}(\tau)}{\Gamma \mid \mathbf{E};\mathbf{L} \mid p \lhd \mathbf{own}_n\, \tau \vdash \mathtt{delete}(n, p) \dashv \emptyset}$$

S-deref
$$\frac{\Gamma \mid \mathbf{E};\mathbf{L} \vdash \tau_1 \circ\!\!-^{\tau} \tau_1' \qquad \text{size}(\tau) = 1}{\Gamma \mid \mathbf{E};\mathbf{L} \mid p \lhd \tau_1 \vdash {}^*p \dashv x.\, p \lhd \tau_1', x \lhd \tau}$$

S-sum-assgn
$$\frac{\overline{\tau}_i = \tau \qquad \tau_1 -\!\!\circ^{\Sigma\overline{\tau}} \tau_1'}{\mathbf{E};\mathbf{L} \mid p_1 \lhd \tau_1, p_2 \lhd \tau \vdash p_1 \overset{\text{inj } i}{:=} p_2 \dashv p_1 \lhd \tau_1'}$$

Fig. 1. A selection of the typing rules of λ_{Rust} (helper judgments and instructions).

Figure 1: Rust's type-system rules formalized in Sequent Calculus

The Dilemma With Established Programming Languages

Safety vs. Control

- **All programming languages strike some opinionated balance between two features: safety and control.**

- Safe languages restrict the developer's expressiveness in favor of eliminating certain classes of bugs.

- Languages with high control allow the developer to work less inhibited, but it also allows you to "shoot yourself in the foot."

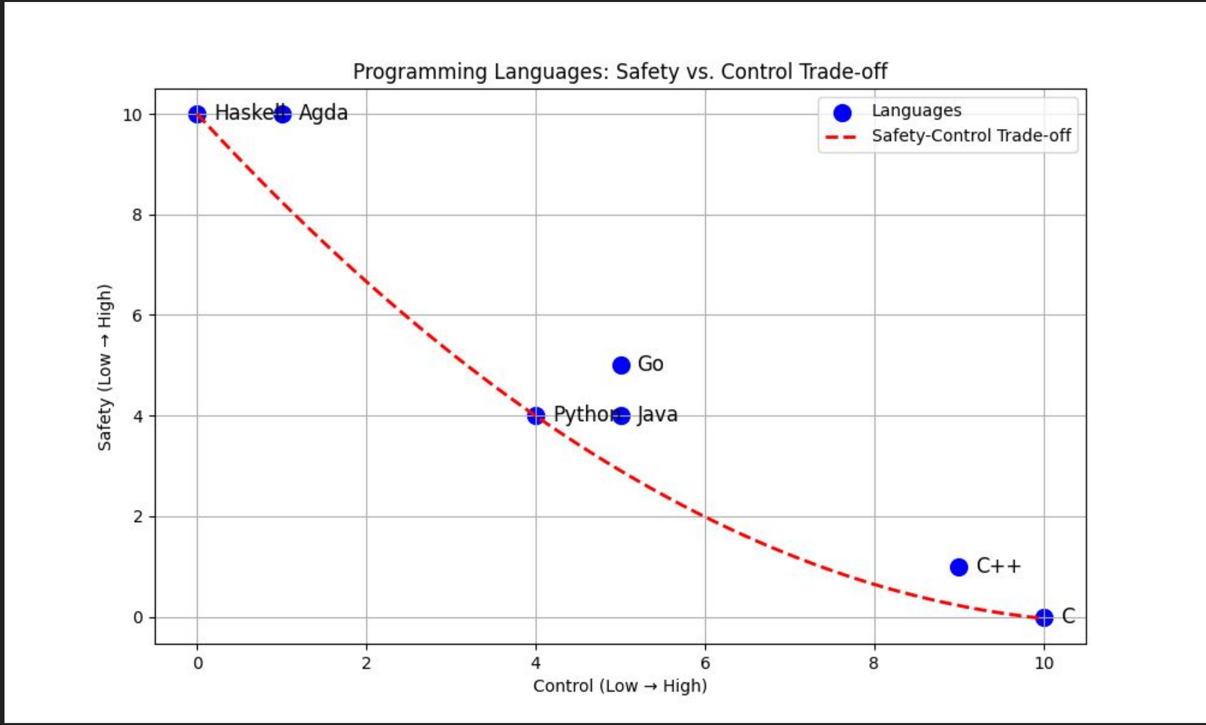

Figure 2: Charting the relationship between developer control + productivity against programming language safety.

An Ideal Programming Language

- An ideal programming language is one that lies at the intersection of maximum safety and maximum control.
  - The user cannot write code that shoots themselves in the foot due to safety checks.
  - The constraints are permissive. They don't interfere with direct control over the machine.
- **This maximizes productivity: less time is spent debugging, but the developer can still write practical code uninhibited.**

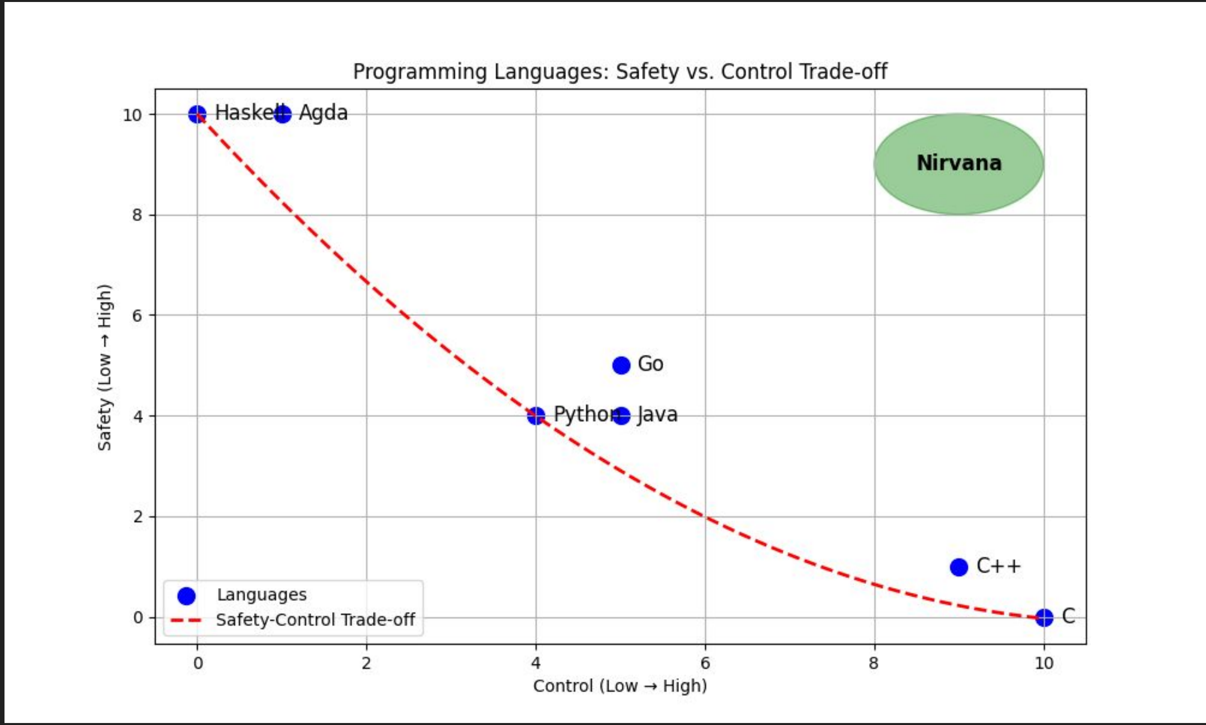

Figure 3: An ideal language would maximize safety *and* control.

Recent Trends In Language Design

- In recent years, programming language research has focused on making practical languages safer, and safe languages more practical.
- Examples include:
  - Typescript on top of JS.
  - Additions to C++: `std::optional` and `std::variant` to avoid null dereferencing and unsafe unions.
  - Wider adoption of functional programming techniques.

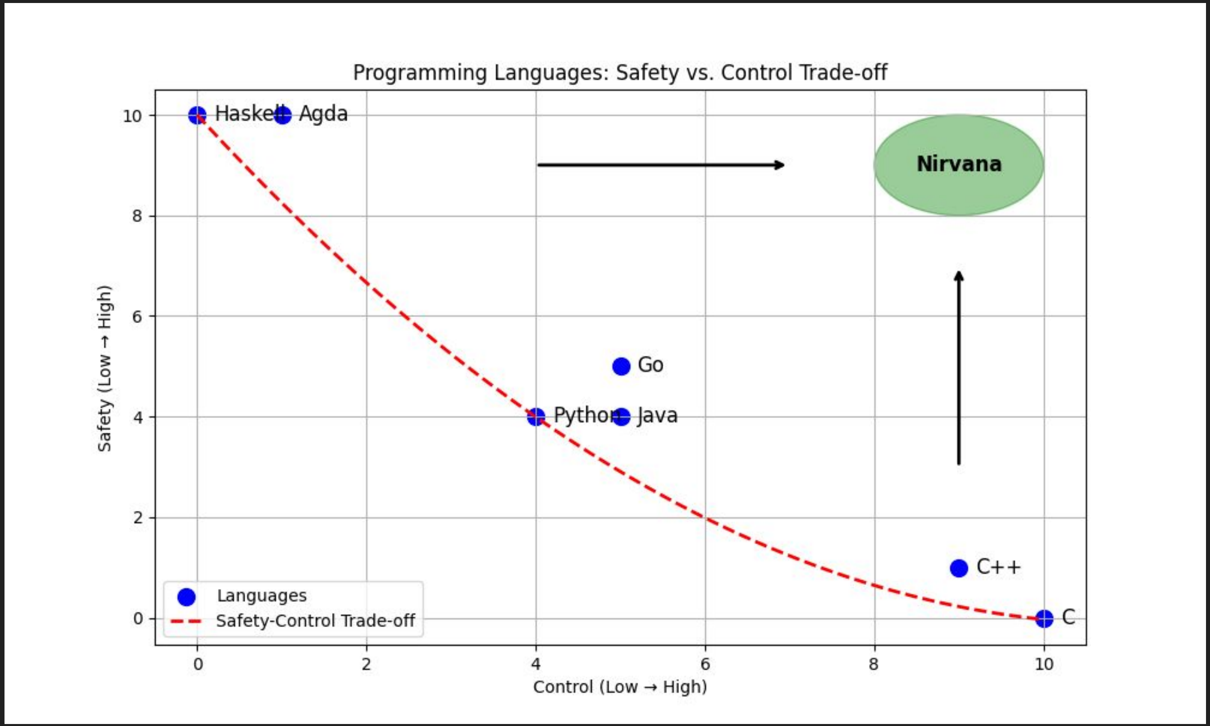

Figure 4: Recent trends show a tendency towards a better balance in safety and control in comparison to older techniques.

Biggest Safety Concern: Memory Management

- Memory safety is the biggest cause of most bugs and security flaws in modern software.
    - **Google and Microsoft report that memory errors compose 70% of their software defects.**
- Traditionally, programming languages prevent invalid memory usage with *runtime* checks:
    - Garbage collection
    - Reference counting
    - Removing pointer semantics from the language entirely
- These strategies come at a high sacrifice to the developer:
    - Large, cumbersome runtimes
    - Slower programs
    - Users can’t access memory directly
        - Unsuitable for embedded, operating systems, etc.

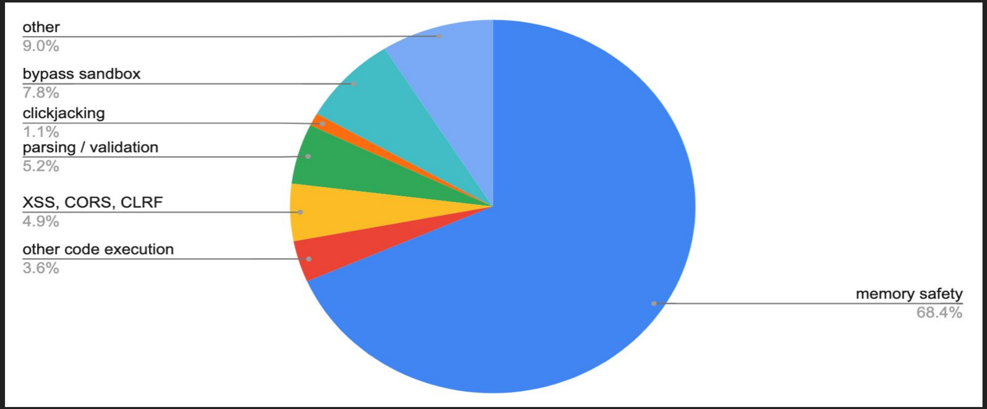

Figure 5: ~70% of all security bugs in Chrome are memory errors, as reported by Google.

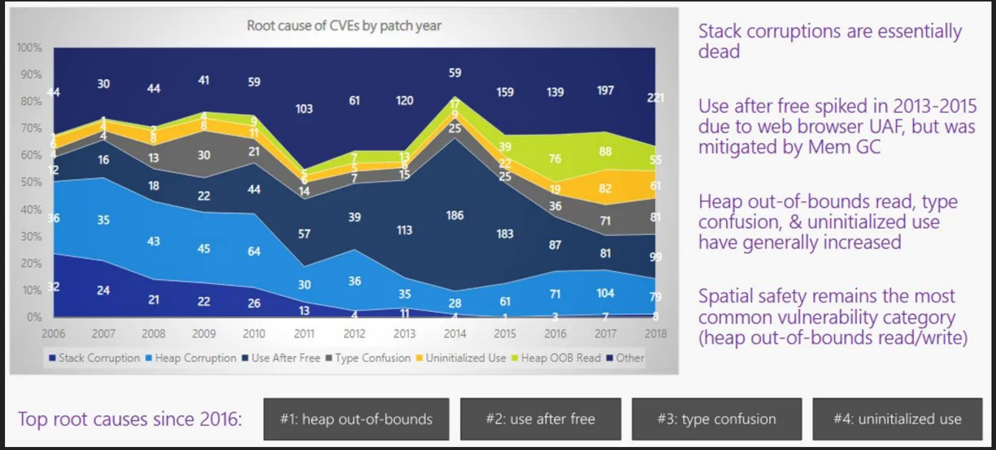

Figure 6: Microsoft finds a similar percentage of memory error contributions to bugs, ~70%

New Techniques For Memory Safety

Rust’s Predecessor: Cyclone

- Cyclone is one of the first attempts at a memory safe dialect of a C-like language.

- Cyclone’s compiler statically deduces how long pointers live, and confirms validity of all pointer accesses.

- ***This comes at a productivity cost:***
    - Pointer lifetimes are always explicitly written by the programmer.
    - Juggles multiple specialized pointer types.
    - Exceptions (`setjmp`, `longjmp`), pointer arithmetic, type casts, and more are disallowed.

- ***Cyclone is safer than C, but much less practical due to the major sacrifices.***

```
trie_t<‘r> trie_lookup(region_t<‘r> r, trie_t<‘r> t,
                       char *‘H buff) {
  switch (t->children) ... // dereferences t
}
int ins_typedef(uregion_key_t<‘r> k,
                trie_t<‘r> t, char *‘H s ; {}) {
  { region<‘r> h = open(k);  // may access ‘r, not k
    trie_t<‘r> t_node = trie_lookup(h,t,s);
    ...
  } // k unconsumed, ‘r inaccessible
  return 0;
}
```

Figure 7: Example code snippet for Cyclone, the precursor to Rust

Enter: The Rust Programming Language

The Rust Programming Language

- Rust is a programming language, built on the research from Cyclone, with a better balance of safety and developer burden.

- **Rust avoids many of the pitfalls of Cyclone:**
    - The same pointer/reference types as C++, *no extra types needed*.

    - Pointer lifetimes are inferred — *little to no developer burden*.

    - Pointer arithmetic, type casts, etc. are allowed, unlike Cyclone.

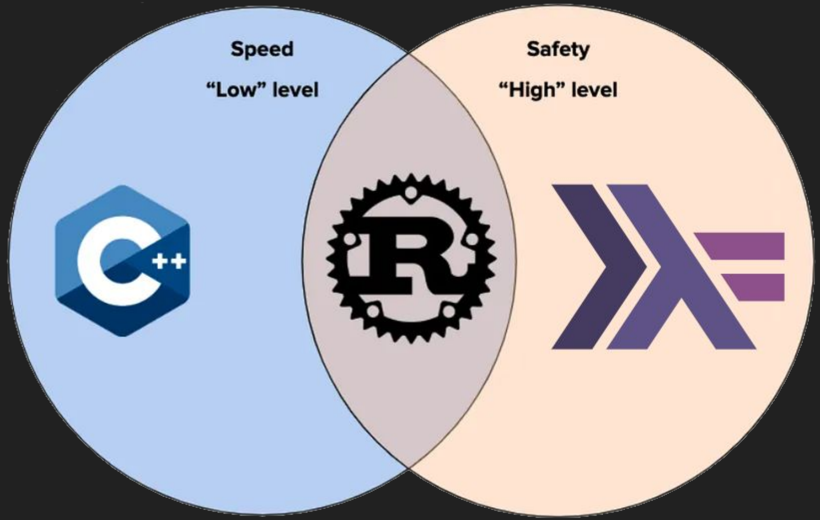

Figure 8: Rust's in comparison to C++ and Haskell's design choices

Rust's Memory Model: Ownership & Borrowing

- Rust's introduces a new strategy for managing memory: the ownership and borrowing model.

- The borrow checker enforces just the following four rules to enable Rust's safety guarantees:
  1. **Each value has a single owner** (one variable owns the data)
  2. **When the value goes out of scope, it is dropped** (memory is freed)
  3. You may have exactly *one mutable reference* **or** *infinite immutable references* to a value at a time
  4. **Live references must always be valid**

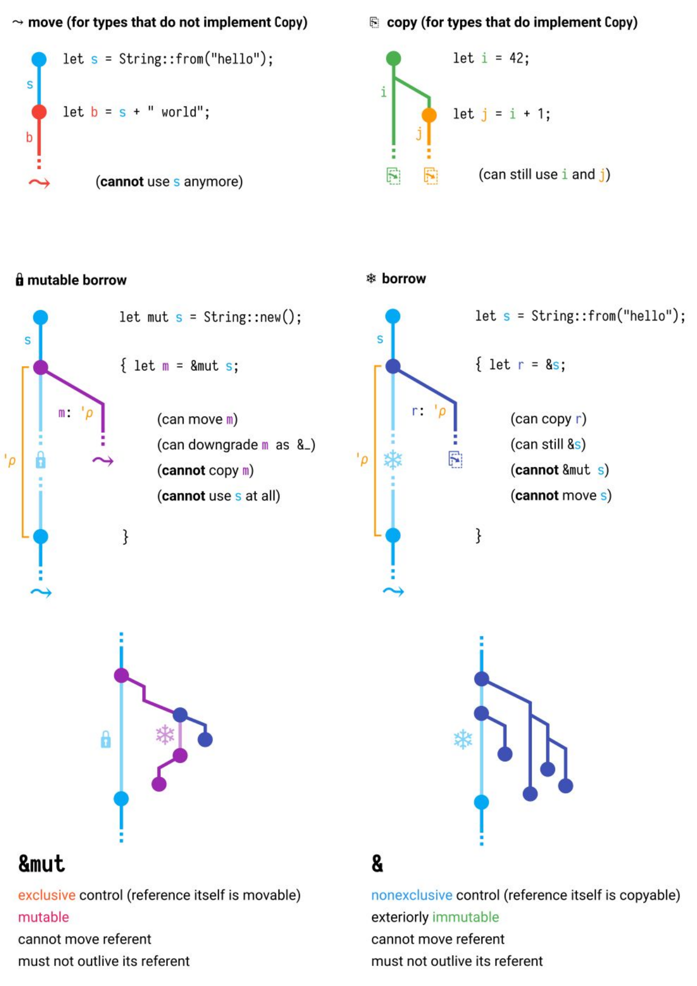

Figure 9: Rust's ownership and borrowing rules

The Benefits Of Ownership & Borrowing

- Rust’s innovative memory model allows developers to access all the *usefulness* of languages like C++, while eliminating memory errors.

- The ownership and borrowing model *statically* guarantees:
  - ***Automatic memory management without GC!***
  - *No use-after-free*
  - *No double-frees*
  - *No null-pointer dereferences*
  - *No thread data races*

- Ownership and borrowing also enables optimizations that are impossible for C++

  - Enforcing a single variable “owner” per value permits powerful strict-aliasing optimizations

```rust
// Lifetimes are annotated below with lines denoting the creation
// and destruction of each variable.
// `i` has the longest lifetime because its scope entirely encloses
// both `borrow1` and `borrow2`. The duration of `borrow1` compared
// to `borrow2` is irrelevant since they are disjoint.
fn main() {
    let i = 3; // Lifetime for `i` starts. ────────────────┐
    //                                                     │
    { //                                                   │
        let borrow1 = &i; // `borrow1` lifetime starts. ──┐│
        //                                                ││
        println!("borrow1: {}", borrow1); //              ││
    } // `borrow1` ends. ─────────────────────────────────┘│
    //                                                     │
    //                                                     │
    { //                                                   │
        let borrow2 = &i; // `borrow2` lifetime starts. ──┐│
        //                                                ││
        println!("borrow2: {}", borrow2); //              ││
    } // `borrow2` ends. ─────────────────────────────────┘│
    //                                                     │
}   // Lifetime ends. ─────────────────────────────────────┘
```

Figure 10: Rust’s borrowing semantics through an example.

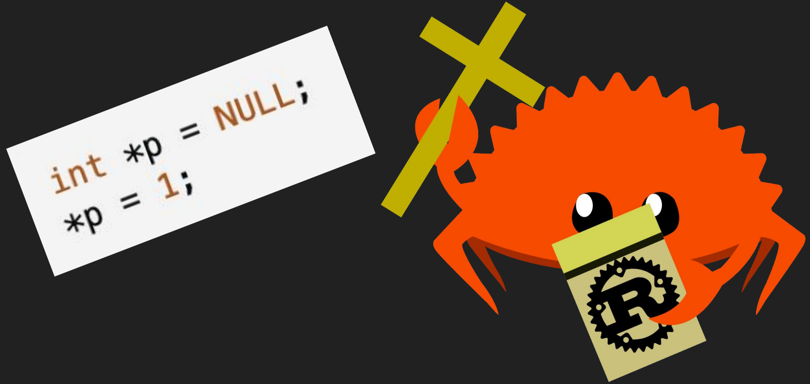

Figure 11: Rust performing an exorcism against access violations.

If Rust is so safe, it must give up expressiveness… right?

A Tour De Force Of Rust’s Expressiveness

Rust Is Expressive Without Compromise

- To the right is an example program that shows some of Rust's expressive power: it's similar to typical C++ code.

- **No safety annotations to be found!** Programmers can focus on what matters.

- Zero-cost abstractions: Rust programs perform on-par with manually managed C++ code.
    - No runtime cost for safety!

```rust
fn main() {
    // Create two points and print them
    let mut p1 = Point::new(3, 4); // `p1` must be mutable to call `shift`
    let p2 = Point::new(5, 6);
    println!("Point #1: {:?}", p1);
    println!("Point #2: {:?}", p2);

    // Move the first point and print it
    p1.shift(1, 1);
    println!("Point #1 after moving: {:?}", p1);

    // Print the distance between the two points
    println!("Distance between p1 and p2: {}", p1.distance_to(&p2));
}

// A point in 2D space
#[derive(Debug)]
struct Point {
    x: i32,
    y: i32
}

impl Point {
    // A constructor for a 2D point
    fn new(x: i32, y: i32) -> Point {
        Point {x, y}
    }

    // Move the point by a given change in X and change in Y.
    // This mutates the current point
    fn shift(&mut self, dx: i32, dy: i32) {
        self.x += dx;
        self.y += dy;
    }

    fn distance_to(&self, other: &Point) -> f64 {
        let dx = self.x - other.x;
        let dy = self.y - other.y;
        ((dx * dx + dy * dy) as f64).sqrt()
    }
}
```

```
Point #1: Point { x: 3, y: 4 }
Point #2: Point { x: 5, y: 6 }
Point #1 after moving: Point { x: 4, y: 5 }
Distance between p1 and p2: 1.4142135623730951
```

Figure 12: An example Rust program implementing a 2D point data-type.

Rust Is Expressive Without Compromise

- Despite its great performance and safety guarantees, it often feels like there's no trade-offs.
- This example converts a custom data-structure to/from JSON, while gracefully handling possible errors.
- An equivalent program in C++ would be more complex, and can't statically generalize for arbitrary custom data-types in the same way.

```rust
use serde::{Deserialize, Serialize};
use serde_json;
use std::error::Error;

/// A struct representing a user profile
#[derive(Debug, Serialize, Deserialize)]
struct User {
    name: String,
    age: u8,
    email: String,
    active: bool,
}

fn main() -> Result<(), Box<dyn Error>> {
    // Sample JSON data
    let json_data = r#"
    {
        "name": "Alice",
        "age": 30,
        "email": "alice@example.com",
        "active": true
    }"#;


    // Deserialize JSON into a Rust struct
    let mut user: User = serde_json::from_str(json_data)?;
    println!("Parsed JSON into Rust struct: {:?}", user);

    // Modify the struct
    user.active = false;
    user.age += 1;

    // Serialize struct back to JSON
    let updated_json = serde_json::to_string_pretty(&user)?;
    println!("\nUpdated JSON:\n{}", updated_json);

    Ok(())
}
```

Figure 13: Convert custom data-types to/from JSON.

```
(dune) /Users/adam/Documents/school/540/testing$ ./testing -h
Parsed JSON into Rust struct: User {
    name: "Alice",
    age: 30,
    email: "alice@example.com",
    active: true,
}

Updated JSON:
{
  "name": "Alice",
  "age": 31,
  "email": "alice@example.com",
  "active": false
}
(dune) /Users/adam/Documents/school/540/testing$
```

Figure 14: Program output for bidirectional JSON conversion.

Rust Is Expressive Without Compromise

- Here, we implement a simple command-line argument parser.
  - Rust auto-generates the `parse` method for our arguments, which are parsed into the fields of our `Args` struct.
- Any user can create and publish their own convenient libraries, just like this one!

```rust
use clap::Parser;

/// A simple program to demonstrate Rust's expressiveness
#[derive(Parser, Debug)]
#[command(version = "1.0", about = "Parses CLI arguments")]
struct Args {
    /// Positional argument: name of the user
    #[arg(default_value = "Stranger")]
    name: String,

    /// A flag to enable verbose mode
    #[arg(short, long)]
    verbose: bool,
}

fn main() {
    let args = Args::parse();

    if args.verbose {
        println!("Verbose mode enabled.");
    }
    println!("Hello, {}!", args.name);
}
```

Figure 15: Parse command line arguments.

```
(dune) /Users/adam/Documents/school/540/testing$ ./testing -h
Parses CLI arguments

Usage: testing [OPTIONS] [NAME]

Arguments:
  [NAME]  Positional argument: name of the user [default: Stranger]

Options:
  -v, --verbose  A flag to enable verbose mode
  -h, --help     Print help
  -V, --version  Print version
(dune) /Users/adam/Documents/school/540/testing$
```

Figure 16: The “help” output of the program.

Overview Of Rust’s Soundness Proof And Sequent Calculus

Explaining The Formal Proof Notation By Example #1

- The notation used in this paper to formally communicate the type system rules is called "**Sequent Calculus**"

- The top of a sequent calculus rule denotes a condition. The top expression implies the bottom expression.

- **Condition #1:** $\Gamma \mid E; L \vdash \kappa$ alive
  - Given a type environment Γ with an expression context E and a lifetime context L, such that the lifetime κ is judged to be "alive"
- **Condition #2:** $\Gamma \mid E; L \vdash \kappa \subseteq \kappa'$
  - The lifetime κ is included in κ' — meaning that the lifetime κ is outlived by κ'
- **Conclusion:** $E; L \vdash \kappa'$ alive
  - If κ is alive in the environment, and κ' outlives κ, then κ' is alive.

If a lifetime κ is alive, and κ' outlives κ, then κ' must also be alive.

LALIVE-INCL

$$\frac{\Gamma \mid \mathbf{E}; \mathbf{L} \vdash \kappa \text{ alive} \qquad \Gamma \mid \mathbf{E}; \mathbf{L} \vdash \kappa \sqsubseteq \kappa'}{\mathbf{E}; \mathbf{L} \vdash \kappa' \text{ alive}}$$

Explaining The Formal Proof Notation By Example #2

- **Condition #1:** $\kappa \subseteq_l \overline{\kappa} \in L$
  - $\boldsymbol{\kappa}$: Denotes an individual lifetime κ, meaning values tied to this lifetime can still be referenced.

  - $\boldsymbol{\kappa \subseteq_l \overline{\kappa}}$: This expresses a *local scope constraint*: the lifetime κ is a local lifetime in $\overline{\kappa}$. **No references tied to $\boldsymbol{\kappa}$ can outlive $\boldsymbol{\overline{\kappa}}$.**

  - $\boldsymbol{\overline{\kappa} \in L}$: The set of lifetimes $\overline{\kappa}$ are tracked in the checker's lifetime context denoted by L.
- **Condition #2:** $\kappa' \in \overline{\kappa}$
  - The lifetime κ' is accessible in $\overline{\kappa}$. **However, this doesn't establish any constraints on $\boldsymbol{\kappa'}$, it may outlive lifetimes in $\boldsymbol{\overline{\kappa}}$.**
- **Conclusion:** $\Gamma \mid E; L \vdash \kappa \subseteq \kappa'$
  - It is provable in this environment that κ lives at most as long as κ'.

References to local variables in a scope cannot outlive their parent scope.

$$\textsc{Lincl-local}\quad \frac{\kappa \sqsubseteq_l \overline{\kappa} \in \mathbf{L} \qquad \kappa' \in \overline{\kappa}}{\Gamma \mid \mathbf{E}; \mathbf{L} \vdash \kappa \sqsubseteq \kappa'}$$

A lot of notation later…

- The entire set of rules is multiple pages long.
- These rules are then transcribed into the **Coq** theorem proving language.
  - If an invalid memory access operation is possible with these rules, then it will be included in the set well-typed programs.
  - The proof assistant confirms that it is impossible to construct use-after-frees, double-frees, null-pointer-dereferences, etc. in the set of well-typed programs.

Rules for lifetimes: $(\Gamma \mid \mathbf{E};\mathbf{L} \vdash \kappa_1 \sqsubseteq \kappa_2$ and $\Gamma \mid \mathbf{E};\mathbf{L} \vdash \kappa$ alive$)$

$$\textsc{Lincl-static}\quad \Gamma \mid \mathbf{E};\mathbf{L} \vdash \kappa \sqsubseteq \mathbf{static}$$

$$\textsc{Lincl-local}\quad \frac{\kappa \sqsubseteq_{\mathrm{l}} \overline{\kappa} \in \mathbf{L} \qquad \kappa' \in \overline{\kappa}}{\Gamma \mid \mathbf{E};\mathbf{L} \vdash \kappa \sqsubseteq \kappa'}$$

$$\textsc{Lincl-extern}\quad \frac{\kappa \sqsubseteq_{\mathrm{e}} \kappa' \in \mathbf{E}}{\Gamma \mid \mathbf{E};\mathbf{L} \vdash \kappa \sqsubseteq \kappa'}$$

$$\textsc{Lincl-refl}\quad \Gamma \mid \mathbf{E};\mathbf{L} \vdash \kappa \sqsubseteq \kappa$$

$$\textsc{Lincl-trans}\quad \frac{\Gamma \mid \mathbf{E};\mathbf{L} \vdash \kappa \sqsubseteq \kappa' \qquad \Gamma \mid \mathbf{E};\mathbf{L} \vdash \kappa' \sqsubseteq \kappa''}{\Gamma \mid \mathbf{E};\mathbf{L} \vdash \kappa \sqsubseteq \kappa''}$$

$$\textsc{Lalive-local}\quad \frac{\kappa \sqsubseteq_{\mathrm{l}} \overline{\kappa} \in \mathbf{L} \qquad \forall i.\, \mathbf{E};\mathbf{L} \vdash \overline{\kappa}_i \text{ alive}}{\Gamma \mid \mathbf{E};\mathbf{L} \vdash \kappa \text{ alive}}$$

$$\textsc{Lalive-incl}\quad \frac{\Gamma \mid \mathbf{E};\mathbf{L} \vdash \kappa \text{ alive} \qquad \Gamma \mid \mathbf{E};\mathbf{L} \vdash \kappa \sqsubseteq \kappa'}{\mathbf{E};\mathbf{L} \vdash \kappa' \text{ alive}}$$

Rules for subtyping and type coercions: $(\Gamma \mid \mathbf{E};\mathbf{L} \vdash \tau_1 \Rightarrow \tau_2$ and $\Gamma \mid \mathbf{E};\mathbf{L} \vdash \mathbf{T}_1 \overset{\mathrm{ctx}}{\Rightarrow} \mathbf{T}_2)$

$$\textsc{T-bor-lft}\quad \frac{\Gamma \mid \mathbf{E};\mathbf{L} \vdash \kappa \sqsubseteq \kappa'}{\Gamma \mid \mathbf{E};\mathbf{L} \vdash \&_{\mu}^{\kappa'} \tau \Rightarrow \&_{\mu}^{\kappa} \tau}$$

$$\textsc{C-subtype}\quad \frac{\Gamma \mid \mathbf{E};\mathbf{L} \vdash \tau \Rightarrow \tau'}{\Gamma \mid \mathbf{E};\mathbf{L} \vdash p \lhd \tau \overset{\mathrm{ctx}}{\Rightarrow} p \lhd \tau'}$$

$$\textsc{C-copy}\quad \frac{\tau \text{ copy}}{\Gamma \mid \mathbf{E};\mathbf{L} \vdash p \lhd \tau \overset{\mathrm{ctx}}{\Rightarrow} p \lhd \tau, p \lhd \tau}$$

$$\textsc{C-split-own}\quad \mathbf{E};\mathbf{L} \vdash p \lhd \mathbf{own}_n\, \tau_1 \times \tau_2 \overset{\mathrm{ctx}}{\Leftrightarrow} p.0 \lhd \mathbf{own}_n\, \tau_1, \lhd\, \mathbf{own}_n\, \tau_2$$

$$\textsc{C-share}\quad \frac{\Gamma \mid \mathbf{E};\mathbf{L} \vdash \kappa \text{ alive}}{\Gamma \mid \mathbf{E};\mathbf{L} \vdash p \lhd \&_{\mathbf{mut}}^{\kappa} \tau \overset{\mathrm{ctx}}{\Rightarrow} p \lhd \&_{\mathbf{shr}}^{\kappa} \tau}$$

$$\textsc{C-borrow}\quad \Gamma \mid \mathbf{E};\mathbf{L} \vdash p \lhd \mathbf{own}_n\, \tau \overset{\mathrm{ctx}}{\Rightarrow} p \lhd \&_{\mathbf{mut}}^{\kappa} \tau, p \lhd^{\dagger\kappa} \mathbf{own}_n\, \tau$$

$$\textsc{C-reborrow}\quad \frac{\Gamma \mid \mathbf{E};\mathbf{L} \vdash \kappa' \sqsubseteq \kappa}{\Gamma \mid \mathbf{E};\mathbf{L} \vdash p \lhd \&_{\mathbf{mut}}^{\kappa} \tau \overset{\mathrm{ctx}}{\Rightarrow} p \lhd \&_{\mathbf{mut}}^{\kappa'} \tau, p \lhd^{\dagger\kappa'} \&_{\mathbf{mut}}^{\kappa} \tau}$$

Rules for reading and writing: $(\Gamma \mid \mathbf{E};\mathbf{L} \vdash \tau_1 \multimapinv^{\tau} \tau_2$ and $\Gamma \mid \mathbf{E};\mathbf{L} \vdash \tau_1 \multimap^{\tau} \tau_2)$

$$\textsc{Tread-own-copy}\quad \frac{\tau \text{ copy}}{\Gamma \mid \mathbf{E};\mathbf{L} \vdash \mathbf{own}_n\, \tau \multimapinv^{\tau} \mathbf{own}_n\, \tau}$$

$$\textsc{Tread-own-move}\quad \frac{n = \mathrm{size}(\tau)}{\Gamma \mid \mathbf{E};\mathbf{L} \vdash \mathbf{own}_m\, \tau \multimapinv^{\tau} \mathbf{own}_m\, \lightning_n}$$

$$\textsc{Tread-bor}\quad \frac{\tau \text{ copy} \qquad \Gamma \mid \mathbf{E};\mathbf{L} \vdash \kappa \text{ alive}}{\Gamma \mid \mathbf{E};\mathbf{L} \vdash \&_{\mu}^{\kappa} \tau \multimapinv^{\tau} \&_{\mu}^{\kappa} \tau}$$

$$\textsc{Twrite-own}\quad \frac{\mathrm{size}(\tau) = \mathrm{size}(\tau')}{\Gamma \mid \mathbf{E};\mathbf{L} \vdash \mathbf{own}_n\, \tau' \multimap^{\tau} \mathbf{own}_n\, \tau}$$

$$\textsc{Twrite-bor}\quad \frac{\Gamma \mid \mathbf{E};\mathbf{L} \vdash \kappa \text{ alive}}{\Gamma \mid \mathbf{E};\mathbf{L} \vdash \&_{\mathbf{mut}}^{\kappa} \tau \multimap^{\tau} \&_{\mathbf{mut}}^{\kappa} \tau}$$

Rules for typing of instructions: $(\Gamma \mid \mathbf{E};\mathbf{L} \mid \mathbf{T} \vdash I \dashv x.\, \mathbf{T}_2)$

$$\textsc{S-num}\quad \Gamma \mid \mathbf{E};\mathbf{L} \mid \emptyset \vdash z \dashv x.\, x \lhd \mathbf{int}$$

$$\textsc{S-nat-leq}\quad \Gamma \mid \mathbf{E};\mathbf{L} \mid p_1 \lhd \mathbf{int}, p_2 \lhd \mathbf{int} \vdash p_1 \leq p_2 \dashv x.\, x \lhd \mathbf{bool}$$

$$\textsc{S-new}\quad \Gamma \mid \mathbf{E};\mathbf{L} \mid \emptyset \vdash \mathtt{new}(n) \dashv x.\, x \lhd \mathbf{own}_n\, \lightning_n$$

$$\textsc{S-delete}\quad \frac{n = \mathrm{size}(\tau)}{\Gamma \mid \mathbf{E};\mathbf{L} \mid p \lhd \mathbf{own}_n\, \tau \vdash \mathtt{delete}(n, p) \dashv \emptyset}$$

$$\textsc{S-deref}\quad \frac{\Gamma \mid \mathbf{E};\mathbf{L} \vdash \tau_1 \multimapinv^{\tau} \tau_1' \qquad \mathrm{size}(\tau) = 1}{\Gamma \mid \mathbf{E};\mathbf{L} \mid p \lhd \tau_1 \vdash {}^{*}p \dashv x.\, p \lhd \tau_1', x \lhd \tau}$$

$$\textsc{S-sum-assgn}\quad \frac{\overline{\tau}_i = \tau \qquad \tau_1 \multimap^{\Sigma\overline{\tau}} \tau_1'}{\mathbf{E};\mathbf{L} \mid p_1 \lhd \tau_1, p_2 \lhd \tau \vdash p_1 \overset{\mathrm{inj}\ i}{:==} p_2 \dashv p_1 \lhd \tau_1'}$$

Fig. 1. A selection of the typing rules of λ_{Rust} (helper judgments and instructions).

Figure 17: One section of the list of type-checking rules enforced in Rust.

Conclusion

- The “ownership and borrowing” memory model pioneered in Rust’s type system is a very important advancement towards increasing developer productivity:
  - This model prevents hard-to-detect bugs at compile time, while maintaining the expressiveness of widely used systems languages: little to no sacrifice.
  - Rust’s memory model is now formally proven.
    - Future languages now have a proven template for statically preventing memory errors and data races.
  - Smart pointer types based on Rust’s type-checking rules could be integrated into the standard libraries of other established systems languages.
- Rust represents a substantial step towards languages with an ideal balance of safety and practicality.

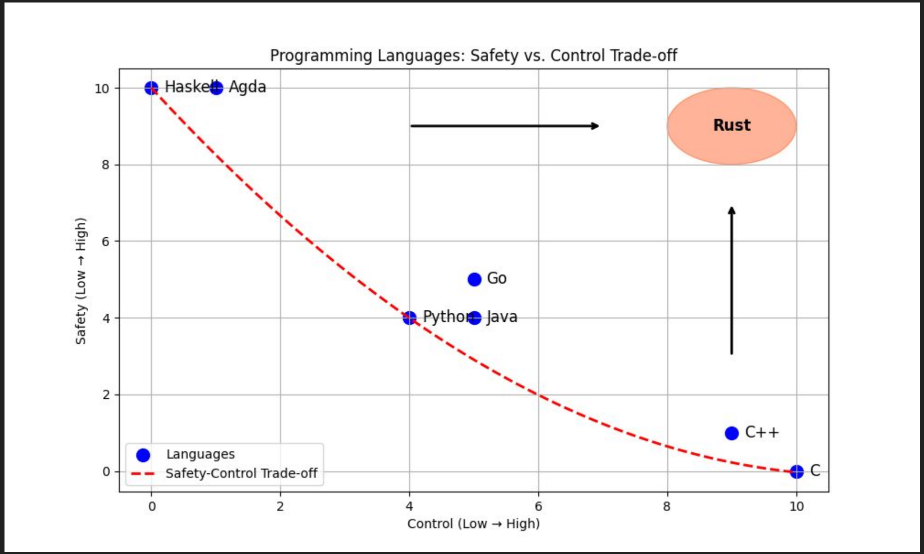

Figure 18: Rust has innovated significant progress towards a safe, practical systems language.

Questions?