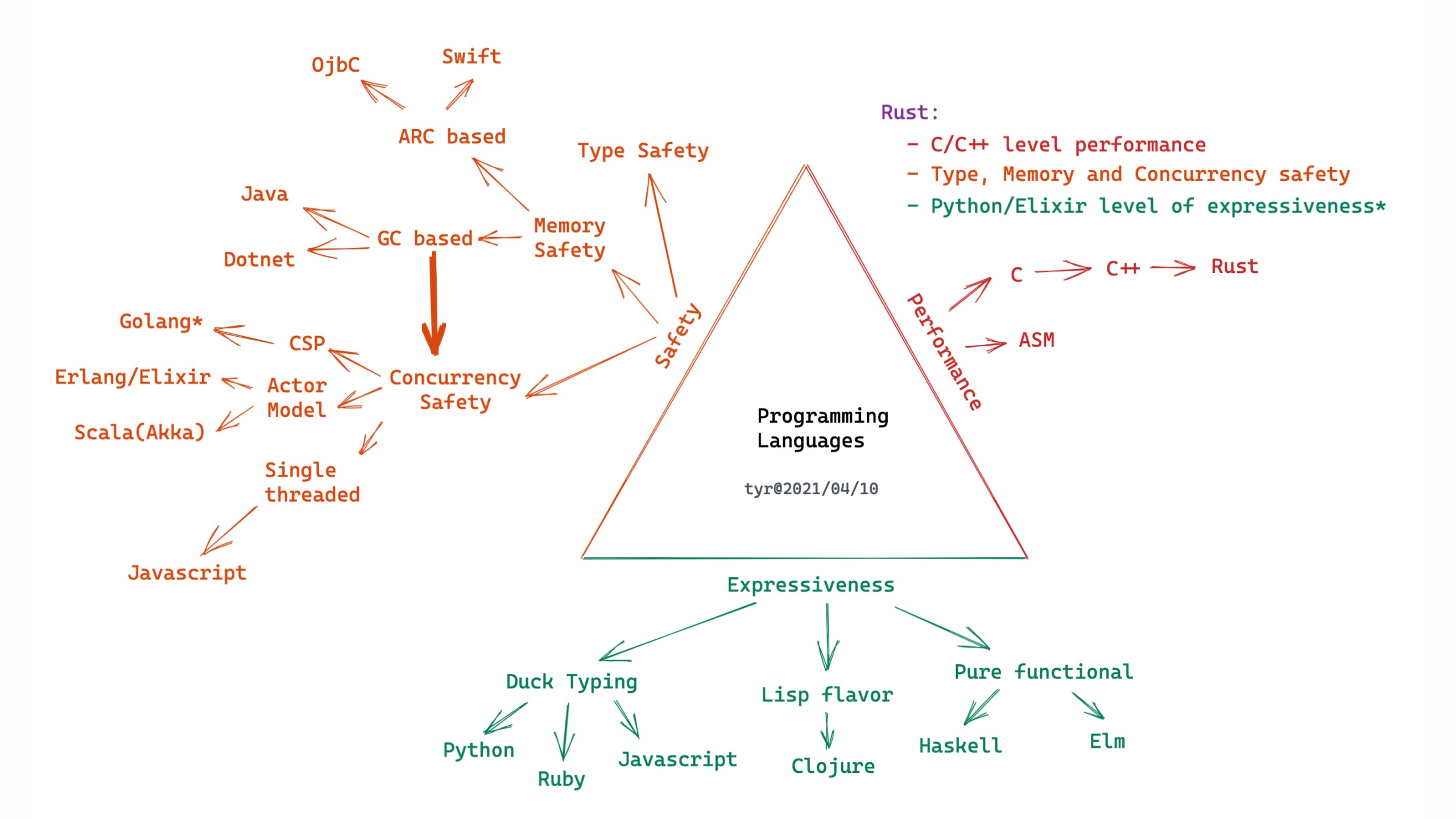
# Rust Trainings All in One

- High-level intro about Rust
- Ownership, borrow check, and lifetime
- Typesystem and data structures
- Concurrency - primitives
- Concurrency - async/await
- Networking and security
- FFI with C/Elixir/Swift/Java
- WASM/WASI
- Rust for real-world problems

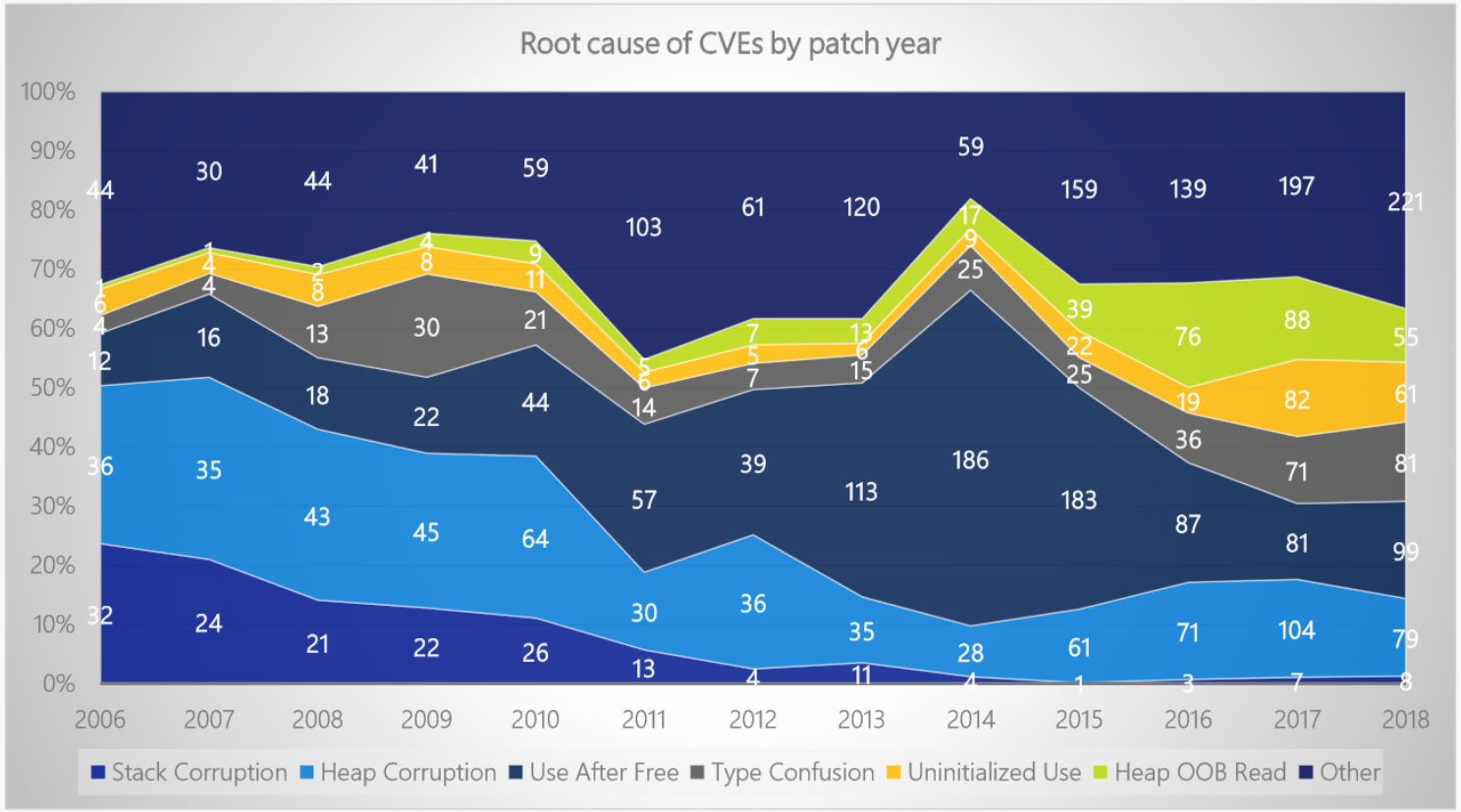
# High-level Intro About Rust

# Why Rust?


OjbC
Swift
ARC based
Java
Dotnet
GC based
Memory Safety
Type Safety
Golang*
CSP
Erlang/Elixir
Actor Model
Scala(Akka)
Concurrency Safety
Single threaded
Javascript
Safety
Programming Languages
tyr@2021/04/10
Performance
C
C++
Rust
ASM
Rust:
- C/C++ level performance
- Type, Memory and Concurrency safety
- Python/Elixir level of expressiveness*
Expressiveness
Duck Typing
Python
Ruby
Javascript
Lisp flavor
Clojure
Pure functional
Haskell
Elm

# Why safety is important?

# Drilling down into root causes




Stack corruptions are essentially dead

Use after free spiked in 2013-2015 due to web browser UAF, but was mitigated by Mem GC

Heap out-of-bounds read, type confusion, & uninitialized use have generally increased

Spatial safety remains the most common vulnerability category (heap out-of-bounds read/write)

Top root causes since 2016:

- #1: heap out-of-bounds
- #2: use after free
- #3: type confusion
- #4: uninitialized use



Note: CVEs may have multiple root causes, so they can be counted in multiple categories

(**Source:** **Memory Safety Issues Are Still the Leading Source of Security Vulnerabilities**)

# Why safety is hard?

- memory safety is not easy (you need to understand the corner cases)
- conccurency safety is really hard (without certain tradeoffs)
- Often you have to bear the extra layer of abstractions
  - normally it means performance hit

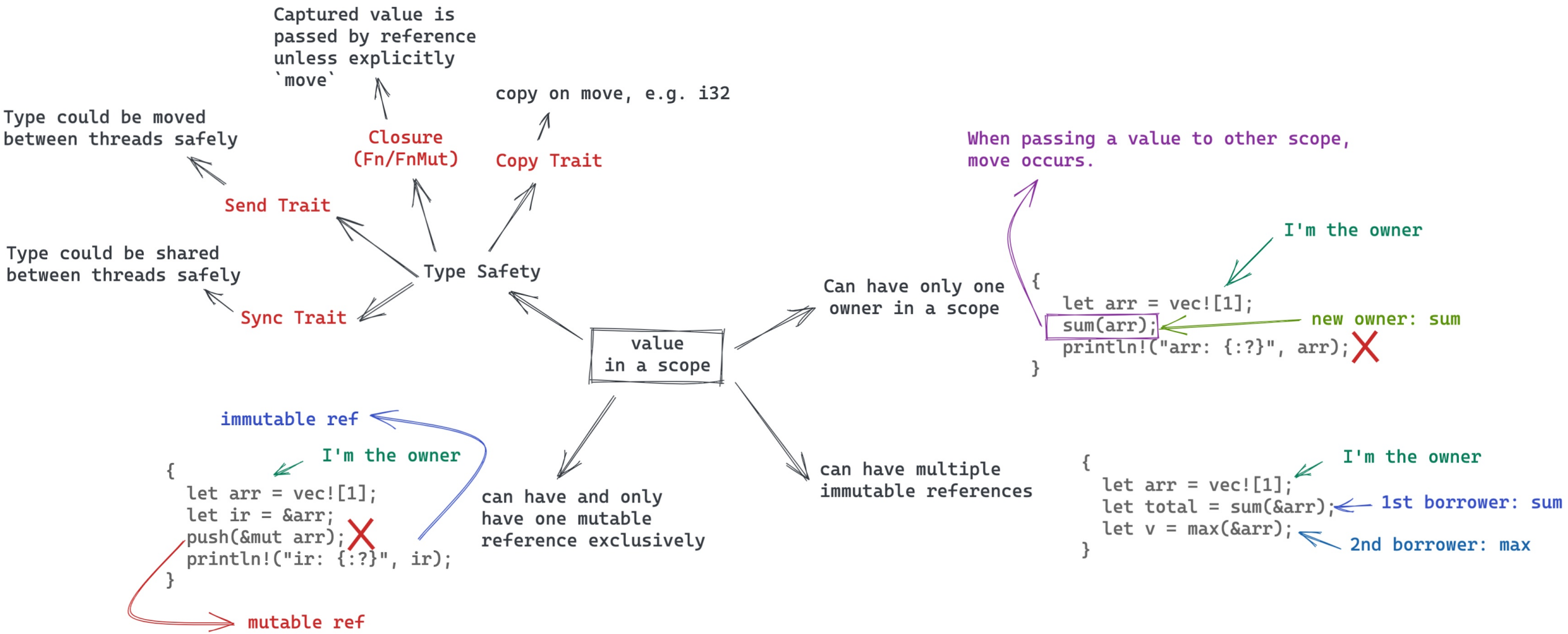
# Memory safety

- Manually - C/C++: painful and error-prone
- Smart Pointers - C++/ObjC/Swift: be aware of cyclical references
- GC - Java/DotNet/Erlang: mubch bigger memory consumption, and STW
- Ownership - Rust: learning curve

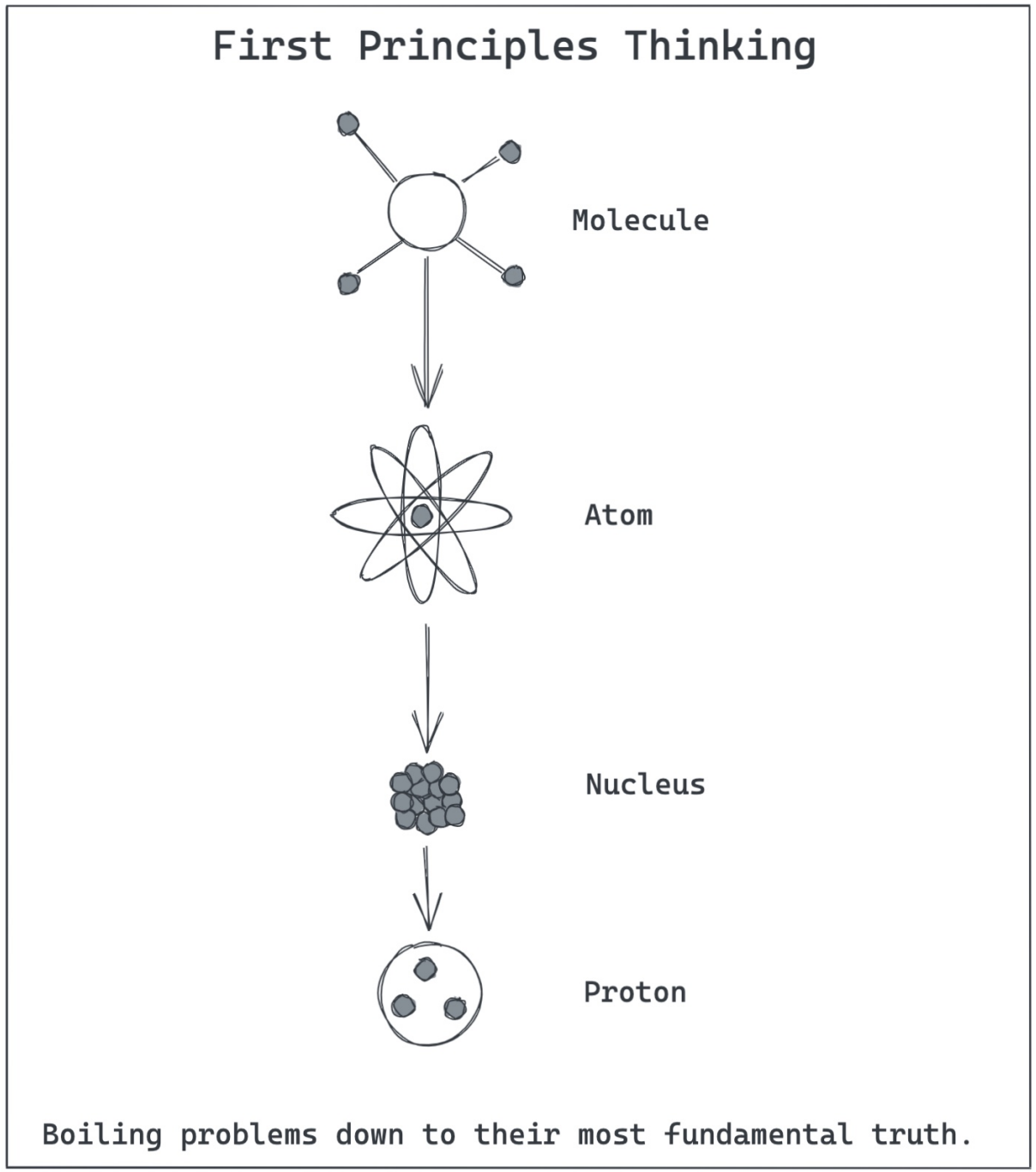
# Concurrency safety

- single-threaded - Javascript: cannot leverage multicore
- GIL - Python/Ruby: multithreading is notorious inefficient
- Actor model - Erlang/Akka: at the cost of memory copy
- CSP - Golang: at the cost of memory copy
- Ownership + Type System - Rust: very elegant and no extra cost!

# How Rust achieve memory safety and conccurency safety?


Captured value is passed by reference unless explicitly `move`
Type could be moved between threads safely
Closure (Fn/FnMut)
copy on move, e.g. i32
Copy Trait
Send Trait
Type could be shared between threads safely
Type Safety
Sync Trait
value in a scope
Can have only one owner in a scope
When passing a value to other scope, move occurs.
I'm the owner
{
let arr = vec![1];
sum(arr);
println!("arr: {:?}", arr);
}
new owner: sum
immutable ref
I'm the owner
{
let arr = vec![1];
let ir = &arr;
push(&mut arr);
println!("ir: {:?}", ir);
}
mutable ref
can have and only have one mutable reference exclusively
can have multiple immutable references
{
I'm the owner
let arr = vec![1];
let total = sum(&arr);
1st borrower: sum
let v = max(&arr);
2nd borrower: max
}

```rust
fn main() {
    let mut arr: Vec<i32> = vec![1, 2, 3];    move occurs because `arr` has type `Vec<i32>`, which does not implement the `Copy` trait
    arr.push(4);

    let result: Result<(), Error> = process(arr);    unused variable: `result`
    let _v: Option<i32> = arr.pop(); // failed since arr is moved    borrow of moved value: `arr`

    // you can have multiple immutable references
    let mut arr1: Vec<i32> = vec![1, 2, 3];
    let ir1: &Vec<i32> = &arr1;
    let ir2: &Vec<i32> = &arr1;

    println!("ir1: {:?} ir2: {:?}", ir1, ir2);

    // but you can't have both mutable and immutable references
    let mr1: &mut Vec<i32> = &mut arr1;    first mutable borrow occurs here
    let mr2: &mut Vec<i32> = &mut arr1;    cannot borrow `arr1` as mutable more than once at a time

    println!("mr1: {:?} mr2: {:?}", mr1, mr2);    first borrow later used here

    // by default, closure borrows the data
    let mut arr2: Vec<i32> = vec![1, 2, 3];
    thread::spawn(|| {    closure may outlive the current function, but it borrows `arr2`, which is owned by the current function
        arr2.push(4);    `arr2` is borrowed here
    });

    // we shall move the data explicitly
    let mut arr3: Vec<i32> = vec![1, 2, 3];
    thread::spawn(move || arr3.push(4));
}

fn thread_safety() {
    // but certain types cannot be moved to other thread safely
    let mut rc1: Rc<Vec<i32>> = Rc::new(vec![1, 2, 3]);
    thread::spawn(move || {    `Rc<Vec<i32>>` cannot be sent between threads safely
        rc1.push(4);
    });
}
```




# Recap

- One and only one owner
- Multiple immutable references
- mutable reference is mutual exclsive
- **use type safety for thread safety**

**With these simple rules, Rust achieved safety with**

**zero cost abstraction**

# Zero Cost Abstraction

## Rust way of searching for solutions

# Let's go to basics about types

- can be used any number of times
  - Other languages: this is how we works
  - Rust: Copy / Clone
- can't be used more than once
  - Other lanugages: ??
  - Rust: move semantics
- must be used at least once
  - Other lanugages: linter will detect that, hopefully
  - `unused_variables`, `unused_assignments`, `unused_must_use`
- must be used exactly once

# References

- The pain of real linear types in Rust
- Substructural type system

# About memory safety

- C/C++

# Ownership, borrow check, and lifetime

# Typesystem and data structures

# Concurrency - primitives

# Concurrency - async/await

# Networking and security

# FFI with C/Elixir/Swift/Java

# WASM/WASI

# Rust for real-world problems

May the Rust be with you