Introduction to Rust for Scientific Computing

Alastair Droop, 2023-06-20

Reproducibility & Scientific Software

(Bio)science is suffering from a major reproducibility crisis

Published results frequently can not be reproduced

A major aspect of this is a lack of reproducibility in scientific software

Many aspects to this problem

- (Modern) scientific code is complex
- Not enough time or resources to “do software engineering properly”
- Not enough training
- Inappropriate tools

The FAIR Principles in Scientific Computing

Findable

- Users can find (specific versions of) the software using a unique and persistent identifier

Accessible

- Software can be accessed and installed using standard tools

Interoperable

- Software adheres to domain-relevant data standards

Reusable

- Software can be run by other users for their specific needs

Applying FAIR Standards to Our Software

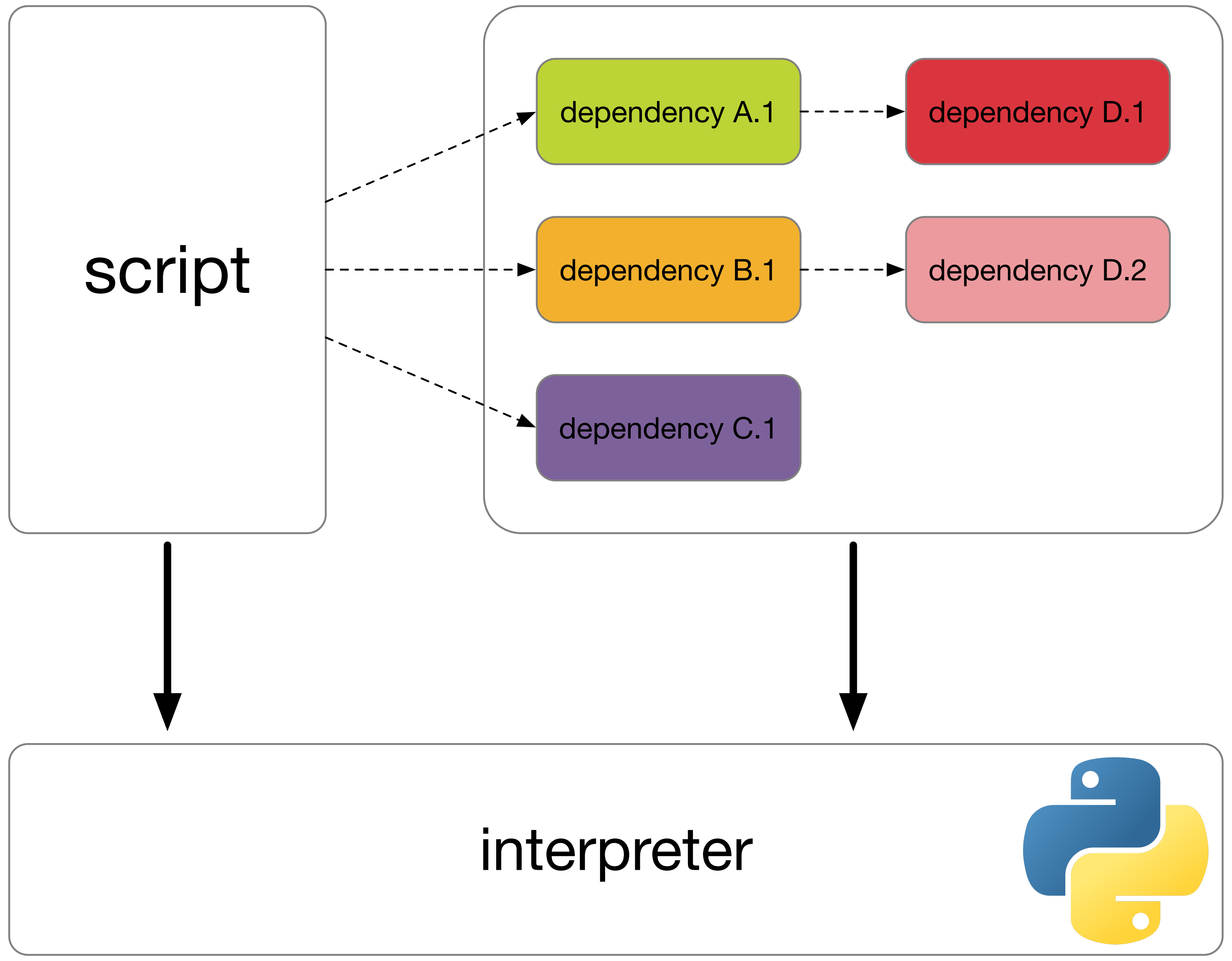
Multiple behaviours are needed to build FAIR software:

- Use of code repositories
- Modularisation & testing
- Continuous Integration
- Coding standards
- High-quality, appropriate documentation
- Semantic Versioning
- Thoughtful containerisation

Many languages and more importantly language stacks make this extremely hard to do in practice

I'm arguing that Rust does many of these things better

Interpreted Languages

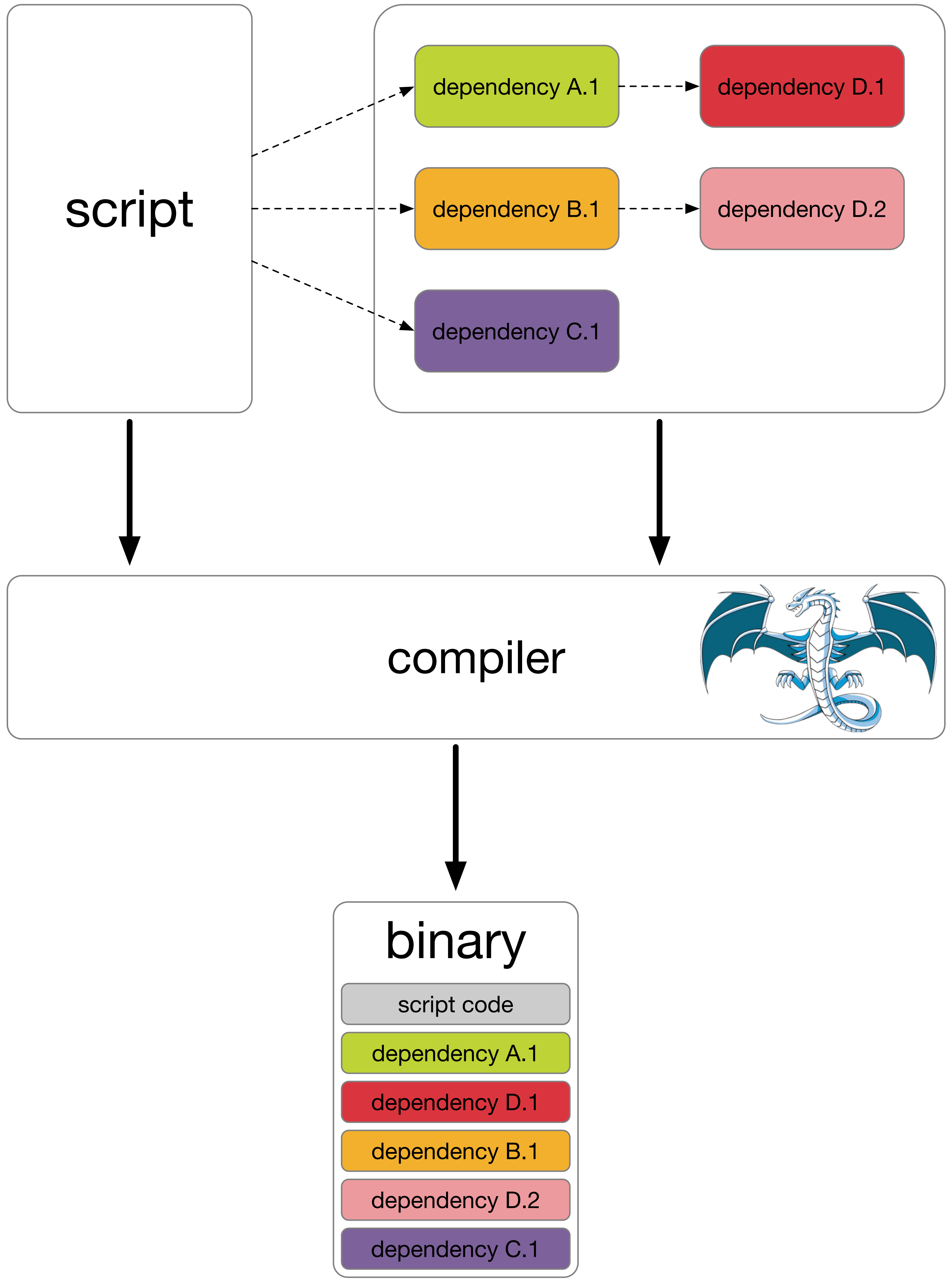


Interpreted languages are executed by an interpreter at runtime, which finds and links dependencies on the fly

+ Code can be almost instantly run (no compilation step)
+ Code can be trivially modified
+ Base script is small
+ Scripts can be platform-agnostic

– Dependencies not included in the code
– Installation can be very complex
– Interpreter needs to be running, so often slower

Static Compilation



Statically-compiled code is executed by the system at runtime, and contains all of its dependencies compiled into a single binary

+ Code does not rely on external packages that can get lost
+ Compiler can run comprehensive checks on the code
+ Installation can be extremely easy
+ No runtime interpreter required
+ Small runtime overhead

– Compilation can be slow
– Executables can be quite large
– Compiled binaries are platform-specific

Dependencies

A fundamental problem of interpreted languages is runtime dependency management

Code needs to locate (at runtime) dependency code and run it

Updating some code that a program depends on can change its behaviour

Virtual environments aim to alleviate this problem

How Should you Install a Python Package?

`setuptools`, **`pip`**, **`venv`**, **`wheel`**, **`twine`**, `pip-tools`, `virtualenvwrapper`, **`pipx`**, `conda`, **`pipenv`**, `poetry`, **`flit`**, **`hatch`**, `pdm`

Most of these are to a greater or lesser degree incompatible

- *Which one do you pick?*
- What happens if you need to install a pipeline with tools that are packaged in an incompatible way?

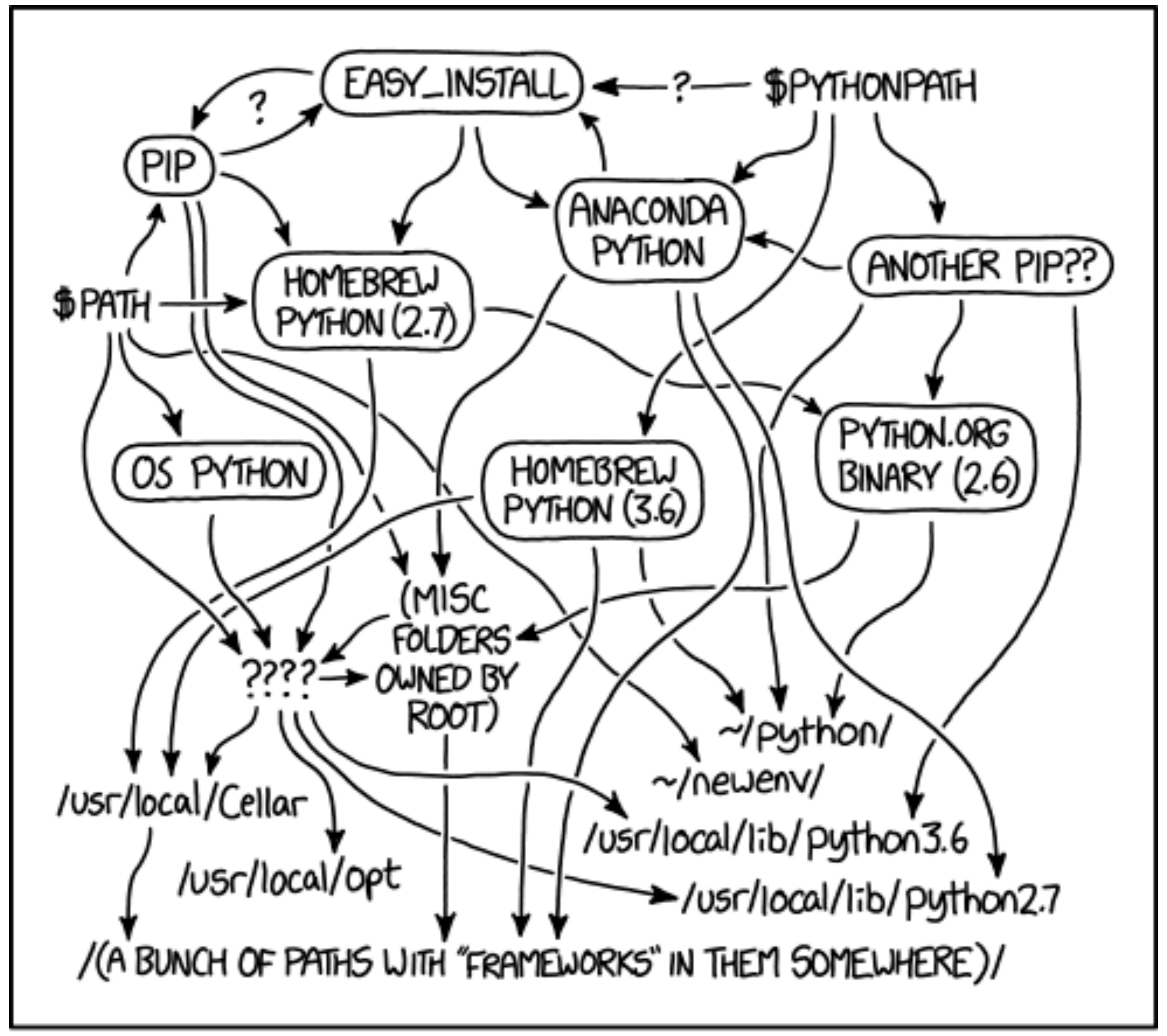
Virtual Environments are Difficult

We often need virtual environments to make R and Python software work

These are directories of packages that are loaded on demand

Virtual Environments are:

- Specific to a Python version
- Trivially updatable (and this is *really bad*)
- Fragile
- Often bloated
- Surprisingly difficult to accurately reproduce
- Difficult for users to set up



MY PYTHON ENVIRONMENT HAS BECOME SO DEGRADED THAT MY LAPTOP HAS BEEN DECLARED A SUPERFUND SITE.

https://xkcd.com/1987

Further Benefits of (Static) Compilation

If we compile our code, the compiler gets a chance to see all the code at once

This allows the compiler to perform rigorous error checking

Rust takes this much further than most languages

+ Many whole classes of bug are no longer possible
+ The need for garbage collection is removed
+ Strict data ownership can be observed, allowing for highly parallel code

“Rust is a modern systems programming language focusing on safety, speed, and concurrency. It accomplishes these goals by being memory safe without using garbage collection.”

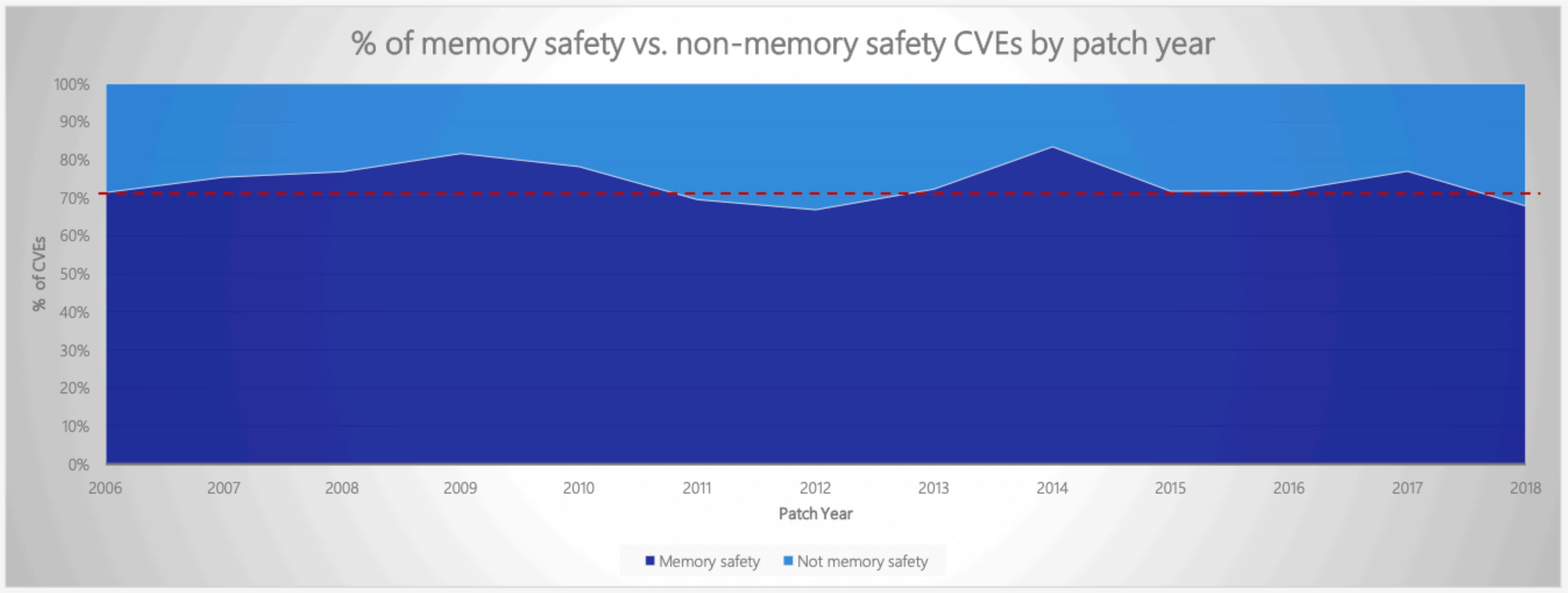
Created in 2006 by Graydon Hoare whilst at Mozilla (used for Mozilla servo)

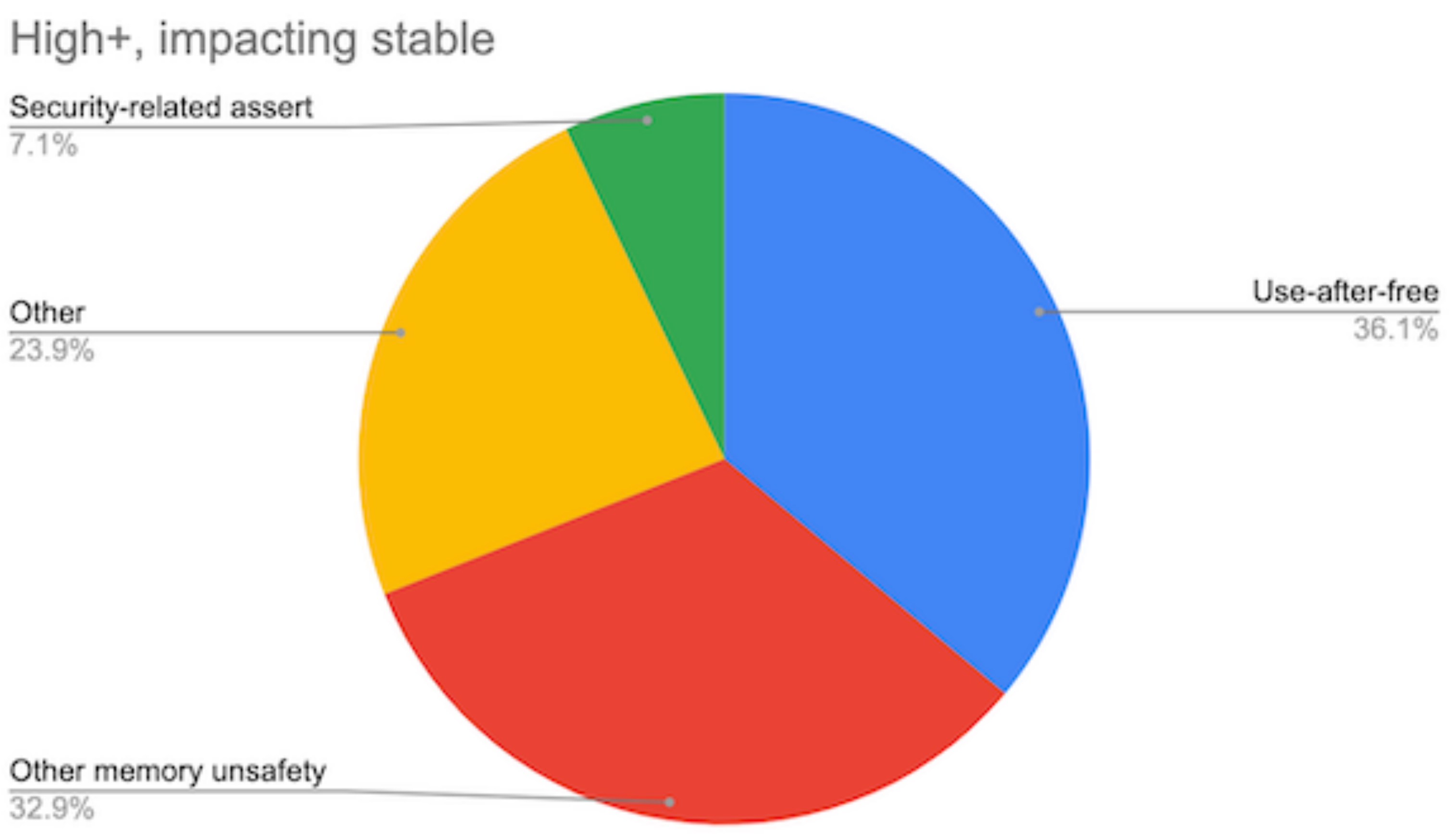
Based strongly on older languages (Nil, Erlang, Limbo, etc…) (“Nothing New”)

Stack Overflow developer survey “most loved language” every year since 2016

Second language adopted for writing the Linux kernel (v6.1, in October 2022)

Memory Safety is A Big Deal





Memory safety bugs account for

- ~70% of all vulnerabilities addressed through a Microsoft security update each year [1]
- ~70% of all serious bugs in the Google Chromium project [2]

Memory safety is a legacy of C (well, ALGOL)'s "Billion dollar mistake" [3]

[1]: https://msrc-blog.microsoft.com/2019/07/18/we-need-a-safer-systems-programming-language/
[2]: https://www.chromium.org/Home/chromium-security/memory-safety
[3]: http://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare

Rust’s Type System

Rust is *statically typed*, so the compiler must know what type al variables are at compile time

- However, the compiler is very good at inferring types, so often you don’t need to worry
- For example, a vector of 64-bit floats would be `Vec<f64>`
- A hash map mapping string keys to 64-bit unsigned integers would be `HashMap<String, u64>`

Collections have powerful & sensible methods that allow “pythonic” manipulations (iterators, *etc*)

Strings (`String`) are totally different to vectors of characters, and are separate to string references (`&str`):

```
let a:String = String::from("Hello, World!");
let b:&str = &a;
```

Generic types allow us to write code that can work for multiple types

- We can set boundaries on which types can be used by specifying which traits a generic type must possess

Ownership

Run-time garbage collection is often used to track application memory

- However, GC is very slow, requires a significant runtime, and vastly complicates parallel software development

Rust uses a completely different approach that makes most memory safety bugs compiler errors:

- Each value has an owner
- There can *only be one owner* at a time
- When the owner goes out of scope, the value is dropped

```
let a = String::from("Hello, World!");
let b = a;
println!("{}", a);
```

References & The Borrow Checker

If we want to refer to a variable without getting ownership of it, we can borrow it:

```
let a = String::from("Hello, World!");
let b = &a;
println!("{}", a);
```

- We can create any number of immutable references to a variable
- We can only ever have a single mutable reference at a time

The Rust compiler makes sure that ownership rules are obeyed. This system is called the “borrow checker”

Borrow checker failures are compiler errors

Structs & Enumerations

Structs allow related values to be packaged together into a meaningful group:

```
struct Read {
    header: String,
    sequence: String,
    quality: Vec<f32>,
}
```

Enums list *all possible* variants of a value:

```
enum MessageResult {
    Success,
    Error(String),
}
```

Pattern Matching

Match expressions branch on a given pattern. The pattern must be complete

- Missing pattern options are compiler errors
- A “catch-all” arm value of “_” is allowed that can cover all other values

```
enum Temperature {
    Kelvin(f32),
    Celsius(f32),
    Fahrenheit(f32),
    Rankine(f32),
}

fn to_absolute(temp: Temperature) -> f32 {
    match temp {
        Temperature::Kelvin(t) => t,
        Temperature::Celsius(t) => t + 273.15_f32,
        Temperature::Fahrenheit(t) => (t + 459.67_f32) * (5_f32 / 9_f32),
    }
}
```

Exception Handling & Optional Values

Without the concept of NULL, how do we represent a failed or empty result?

- The standard library provides two enums to help here: `Option<T>` and `Result<T, E>`

The `Option<T>` represents either some value (of generic type T), or None:

```
enum Option<T> {
    None,
    Some(T),
}
```

`Result<T, E>` enum represents either a success value (of generic type T) or a failure (of generic type E):

```
enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

Cargo.toml

Specifies a code version using semantic versioning

Specifies Rust version

- Once stabilised, features are guaranteed to always be supported
- Dependencies can have different editions to the main code

All dependencies (and their versions) listed

```
[package]
name = "md5walk"
version = "1.0.a"
edition = "2021"

[dependencies]
log = "0.4.14"              ✅
structopt = "0.3.25"        ✅
stderrlog = "0.5.1"         ✅
simple-eyre = "0.3.1"       ✅
rust-crypto = "0.2.36"      ✅
rayon = "1.5.1"             ✅
jwalk = "0.6.0"             ❌ 0.8.1
```

Cargo

Cargo is the rust package manager

Single toolchain for management, compilation, testing, linting & publishing Rust code:

- Creates new code directories `cargo init`
- Adds dependencies `cargo add`
- Runs code linting tools `cargo clippy`
- Standardises code formatting `cargo fmt`
- Runs the compilation `cargo build`
- Installs code `cargo install`

Rust packages are published to crates.io

Testing & Documenting Rust is Easy

Documentation generated using "///"

Examples containing code will be run as tests

Tests can live with the code being tested

Command	Action
`cargo build`	Compile the current crate
`cargo run`	Run the current crate
`cargo clippy`	Run the "Clippy" linter
`cargo fmt`	Reformat all code into a standard style
`cargo test`	Run all tests
`cargo doc`	Compile the crate documentation

```rust
/// Temperature scales
///
/// Allows different temperature scales to be used without confusion
/// See [Wikipedia](https://en.wikipedia.org/wiki/Temperature).
///
pub enum Temperature {
    Kelvin(f32),
    Celsius(f32),
    Fahrenheit(f32),
    Rankine(f32),
}

/// Converts temperatures to an absolute value in Kelvin
/// # Examples
///
/// ```
/// # use example_rust::*;
/// assert_eq!(to_absolute(Temperature::Celsius(0_f32)), 273.15_f32);
/// assert_eq!(to_absolute(Temperature::Celsius(100_f32)), 373.15_f32);
/// ```
pub fn to_absolute(temp: Temperature) -> f32 {
    match temp {
        Temperature::Kelvin(t) => t,
        Temperature::Celsius(t) => t + 273.15_f32,
        Temperature::Fahrenheit(t) => (t + 459.67_f32) * (5_f32 / 9_f32),
        Temperature::Rankine(t) => t * (5_f32 / 9_f32),
    }
}

#[cfg(test)]
mod tests {
    use super::*;
    #[test]
    fn test_zero() {
        assert_eq!(to_absolute(Temperature::Kelvin(0_f32)), 0_f32);
        assert_eq!(to_absolute(Temperature::Celsius(-273.15_f32)), 0_f32);
        assert_eq!(to_absolute(Temperature::Fahrenheit(-459.67_f32)), 0_f32);
        assert_eq!(to_absolute(Temperature::Rankine(0_f32)), 0_f32);
    }
}
```

```rust
/// Converts temperatures to an absolute value in Kelvin
/// # Examples
///
/// ```
/// # use example_rust::*;
/// assert_eq!(to_absolute(Temperature::Celsius(0_f32)), 273.15_f32);
/// assert_eq!(to_absolute(Temperature::Celsius(100_f32)), 373.15_f32);
/// ```
pub fn to_absolute(temp: Temperature) -> f32 {
    match temp {
        Temperature::Kelvin(t) => t,
        Temperature::Celsius(t) => t + 273.15_f32,
        Temperature::Fahrenheit(t) => (t + 459.67_f32) * (5_f32 / 9_f32),
        Temperature::Rankine(t) => t * (5_f32 / 9_f32),
    }
}
```

```
apd500@UOY22M075 ~/D/W/T/2/c/e/src (main)> cargo test
    Finished test [unoptimized + debuginfo] target(s) in 0.01s
     Running unittests src/lib.rs (/Users/apd500/Documents/Work/Talks/2023-rust-intro/code/example-rust/target/debug/deps/example_rust-4002fcc298312
60a)

running 1 test
test tests::test_zero ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.00s

     Running unittests src/bin/docs.rs (/Users/apd500/Documents/Work/Talks/2023-rust-intro/code/example-rust/target/debug/deps/docs-53667294f95132cf
)

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.00s

   Doc-tests example-rust

running 1 test
test src/lib.rs - to_absolute (line 16) ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.38s
```

In example_rust

Enums

Temperature

Functions

to_absolute

Click or press 'S' to search, '?' for more options…

Function example_rust::to_absolute

source · [−]

```
pub fn to_absolute(temp: Temperature) -> f32
```

[−] Converts temperatures to an absolute value in Kelvin

Examples

```
assert_eq!(to_absolute(Temperature::Celsius(0_f32)), 273.15_f32);
assert_eq!(to_absolute(Temperature::Celsius(100_f32)), 373.15_f32);
```

Summary

Rust has a few features that prevent memory bugs and help ensure reliable & reproducible code

- Values are statically typed (this is good)
- Strict ownership model prevents memory errors and data races
- Asynchronous & parallel code is easy & safe to write
- Simple testing and documentation
- Strict error handling
- Unsafe code (for example FFI) is clearly marked with the `unsafe` keyword
- Zero-cost high-level abstractions (*e.g*. iterators) feel like Python
- Code is statically compiled, removing most(?) dependency issues
- Code is semantically versioned
- Cross compilation is easy
- Installation is usually trivial

Useful Links

The Rust homepage	https://www.rust-lang.org/
The Rust Book	https://doc.rust-lang.org/book/
The Cargo Book	https://doc.rust-lang.org/cargo/
The Rustlings Course	https://github.com/rust-lang/rustlings/
Rust by Example	https://doc.rust-lang.org/stable/rust-by-example/
The Rust Standard Library	https://doc.rust-lang.org/std/
The Rust community crate registry	https://crates.io/

The TF Data Science Group




We’re Part of the York University Bioscience Technology Facility

- Bioinformatics
- Machine learning & AI
- Mathematical / statistical techniques
- Modelling, simulation & visualisation
- Data reproducibility, security, anonymity
- Scientific software development

Teaching is increasingly important

- Specific techniques & skills
- Drop-in & “clinic” style advice
- Research computing skills
- Data handling
- Programming

2023 Training Courses

https://www.york.ac.uk/biology/technology-facility/tfcourses/data-science-courses/

Many courses that we can deliver if there is interest:

- Introduction to Neural Networks in Python
- Introduction to Rust programming
- Advanced Rust Programming
- RNA Sequence analysis in R
- Introduction Genome Assembly
- Introduction to Data Science
- Introduction to Research Computing

https://forms.gle/i9pKR33VvGBKvwfh9

alastair.droop@york.ac.uk

@tf_datascience