Rust for high level applications

Lise Henry

Who am I

- Elisabeth Henry
  - A.k.a Lizzie Crowdagger
- Computer science background
- Semi-professional fantasy writer
- I like Rust, but not really into systems programming
  - $\rightarrow$ Crowbook
- A “bad” choice to use Rust for this?

High-level applications

- CLI apps, GUI apps, small games, ...
- Performance is not that important
- Safety is not really a concern
- No need for low-level stuff
- Comparison: Java, Python, Ruby, ...

Rust

- Systems programming
- Blazingly fast
- Prevents segfault
- Safety
- Comparison: C/C++

Doesn't seem like a good fit

```
error[E0499]: cannot borrow `v` as mutable more than once at a time
 --in> test_mut.rs:5:13
  |
3 |     for x in &mut v {
  |                  - first mutable borrow occurs here
4 |         if *x % 2 == 0 {
5 |             v.push(*x + 1);
  |             ^ second mutable borrow occurs here
6 |         }
7 |     }
  |     - first borrow ends here

error: aborting due to previous error
```

Rust is painful

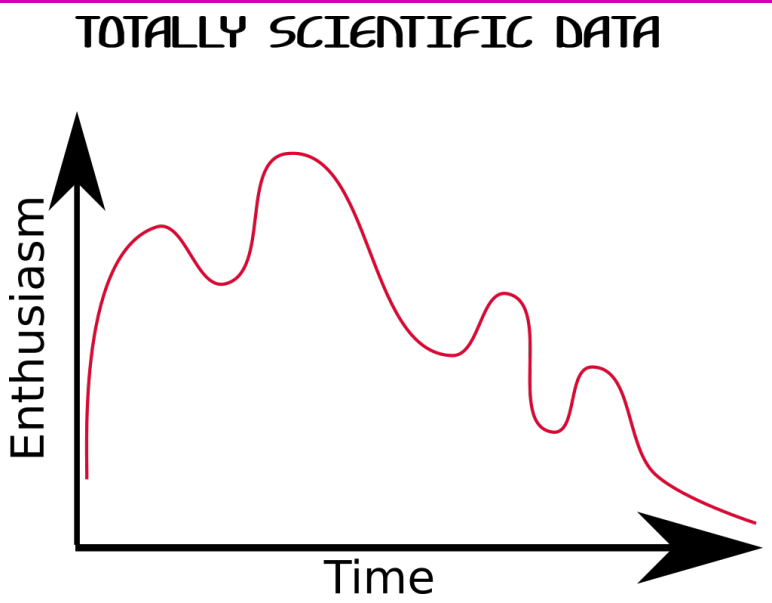
Past experiences with programming

Disclaimer: I am not a great programmer

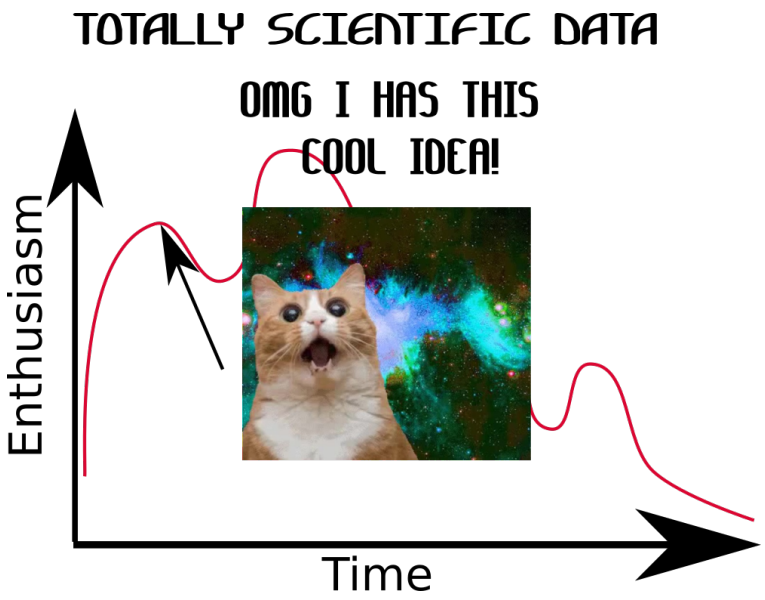
- Lazy
- Not great designer
- Not enough checks and tests
- Easy way instead of the right way

Portfolio of abandoned projects as soon as they grow a bit

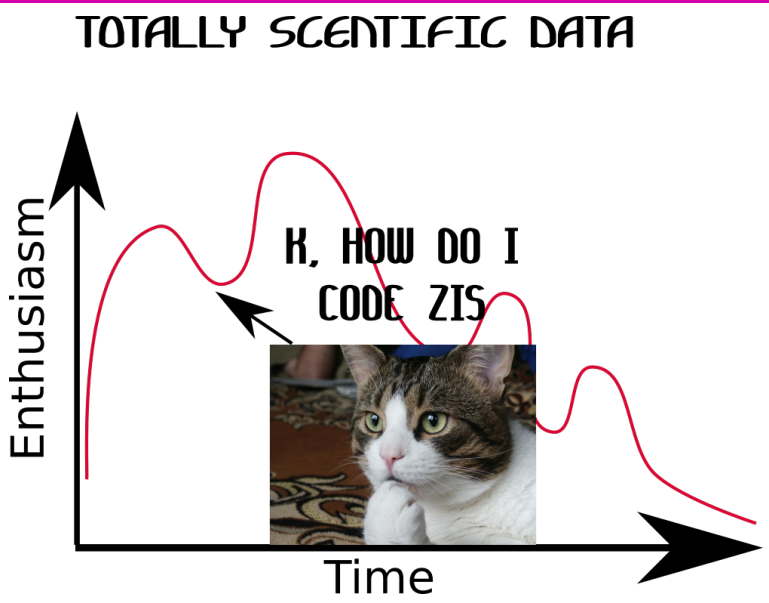
Past experiences with programming



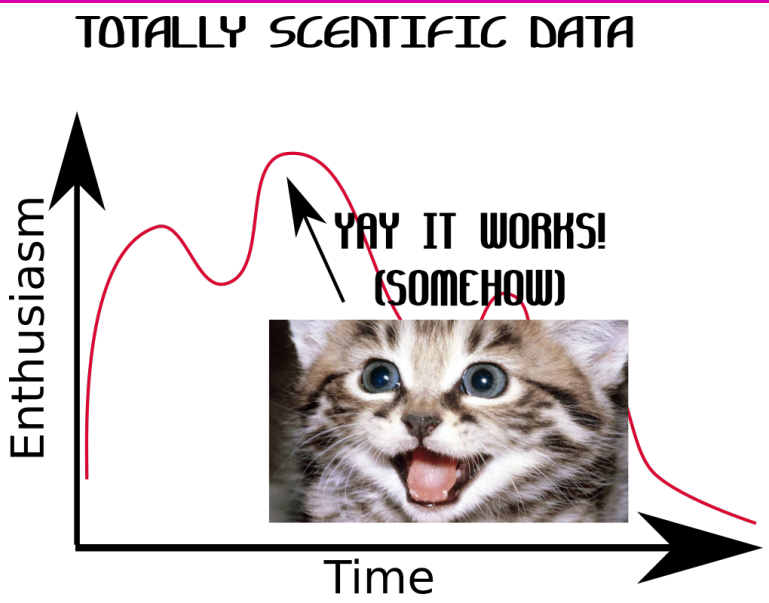
Past experiences with programming



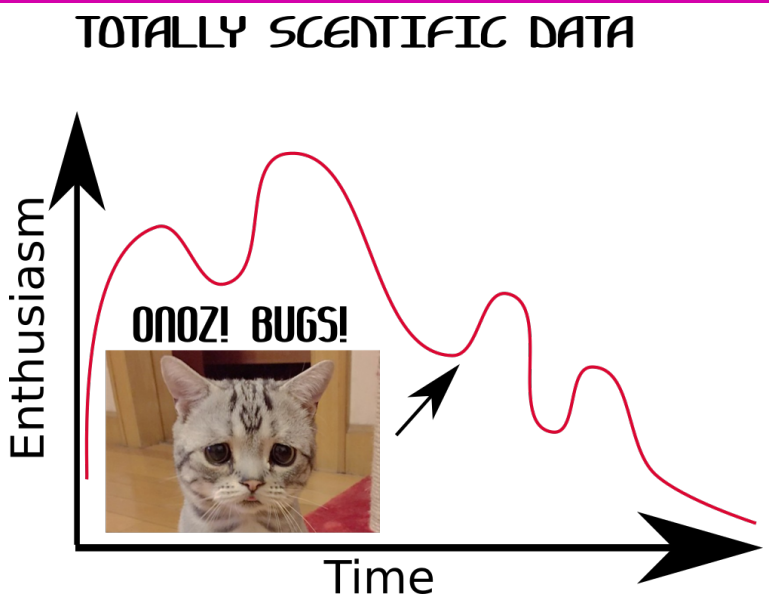
Past experiences with programming



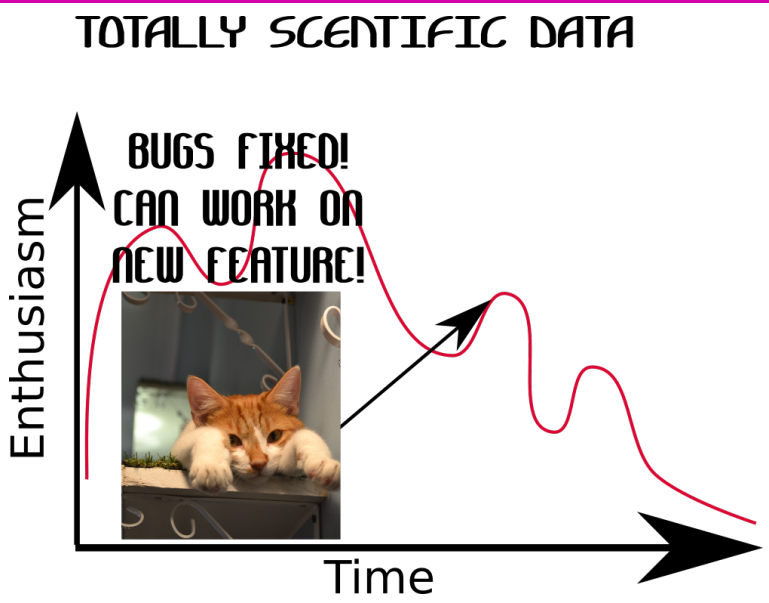
Past experiences with programming



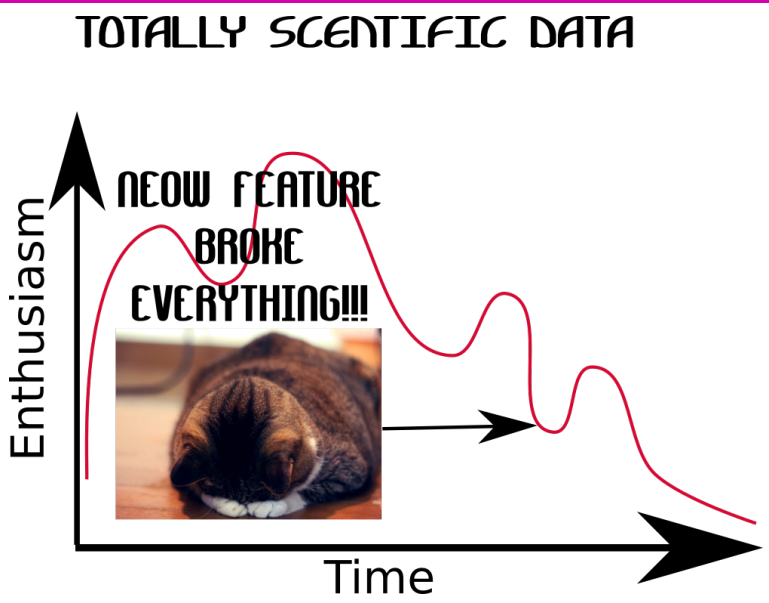
Past experiences with programming



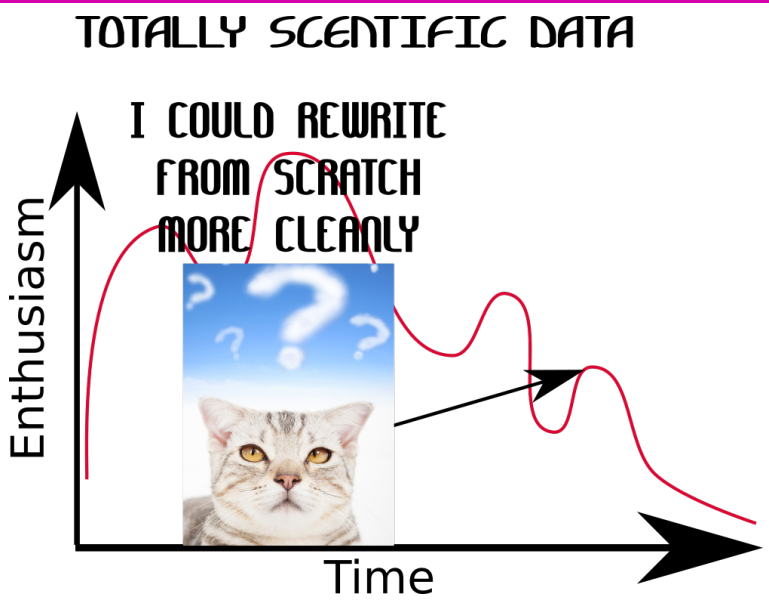
Past experiences with programming



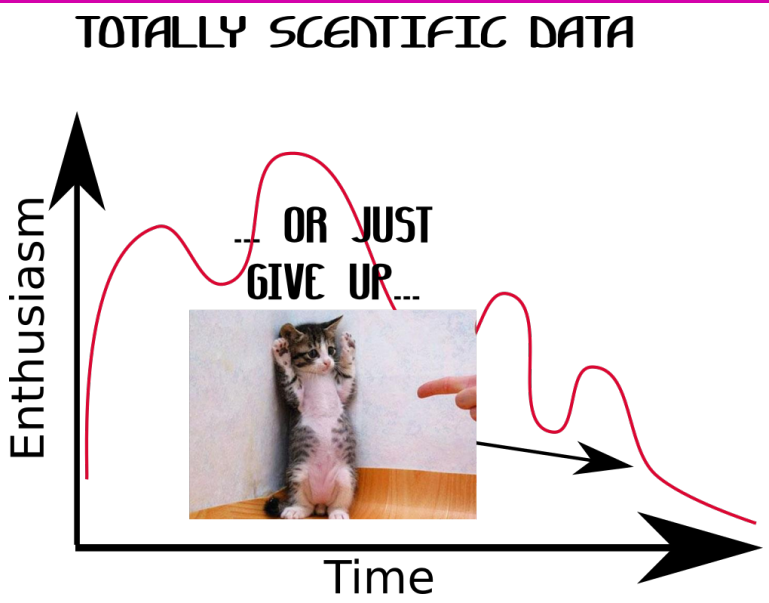
Past experiences with programming



Past experiences with programming



Past experiences with programming



Imperative programming

- Many languages make it easy to start coding
- But also to do things the wrong way
- Complexity
- Hard to reason about
- Solutions:
  - *Good* programmers don't to that
  - It's because you have a bad design

Hiding complexity

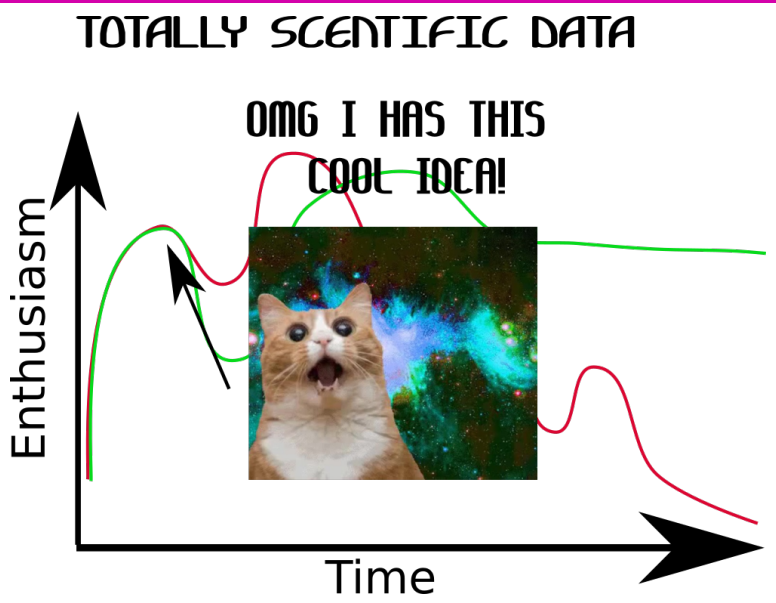
- Lots of easy-to-learn languages hide a lot of complexity
- But that doesn't always make things simpler in the long term

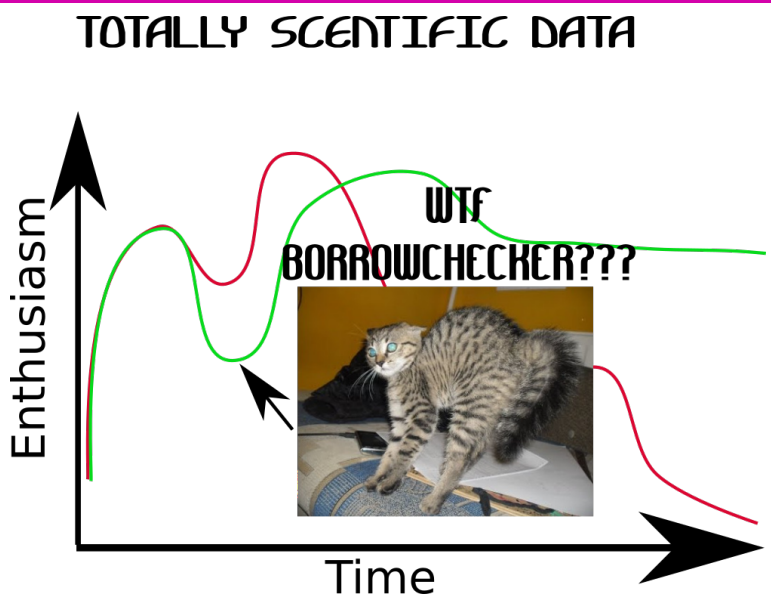
```
a = [1, 2, 3]
b = a
b.append(4)
print a
```

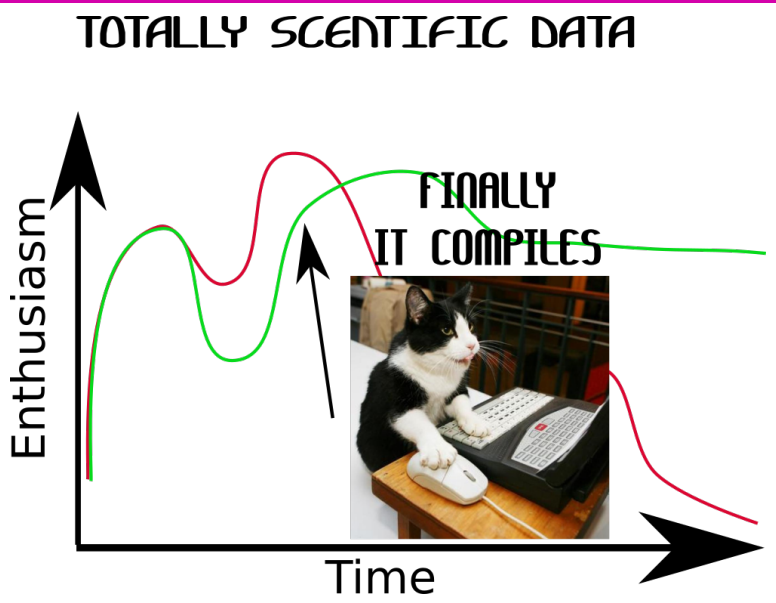
- A bit like hiding pain
- … but ignoring pain isn't always a great idea

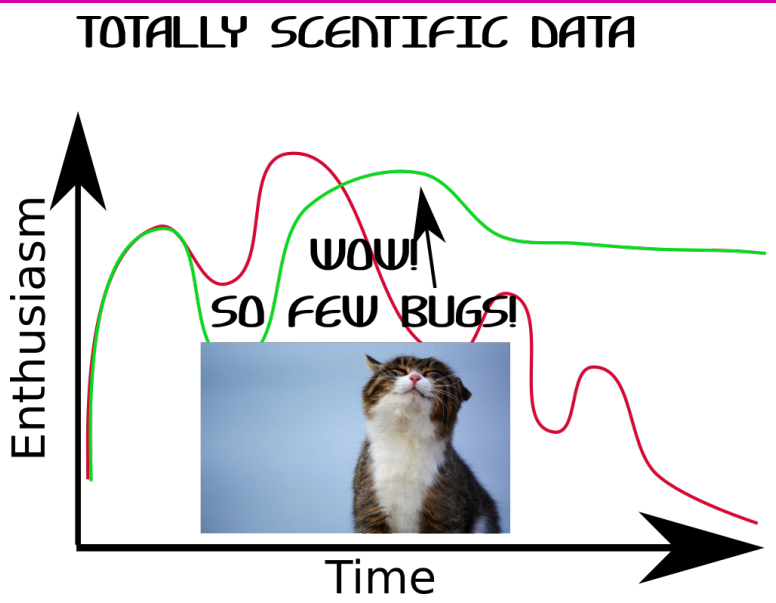
Rust goes a long way to make sure you brush your teeth

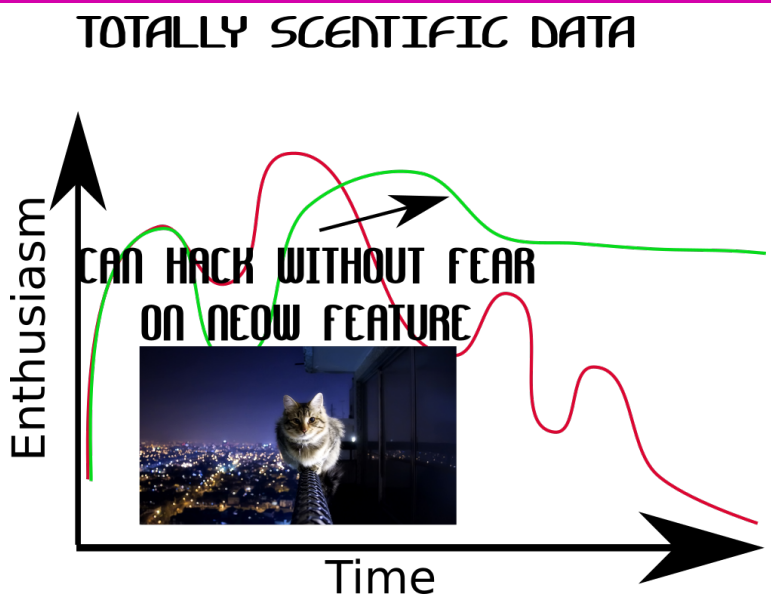
- Lots of checks at compile time
- Type system
  - Enums
  - Traits
- Libraries can leverage all this
  - Make sure you do the right thing

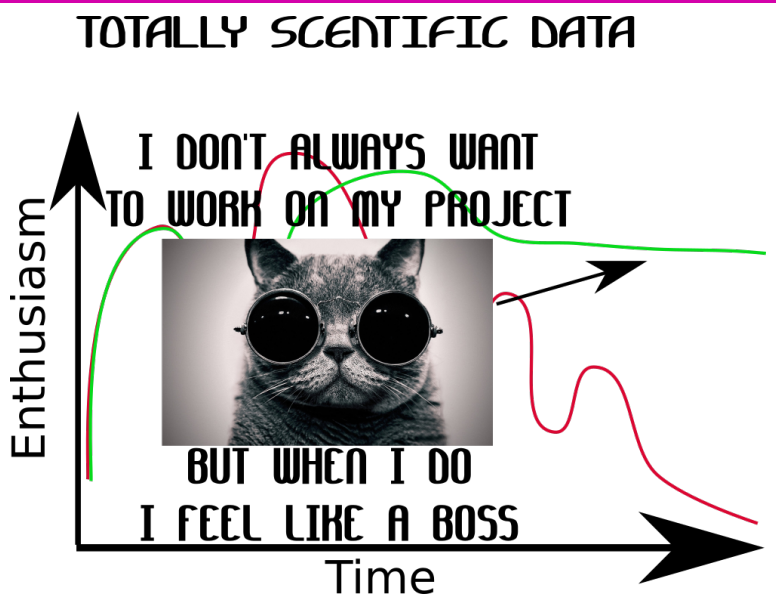

TOTALLY SCENTIFIC DATA
OMG I HAS THIS COOL IDEA!
Enthusiasm
Time


TOTALLY SCENTIFIC DATA
Enthusiasm
Time
WTF
BORROWCHECKER???


TOTALLY SCENTIFIC DATA
FINALLY
IT COMPILES
Enthusiasm
Time


TOTALLY SCENTIFIC DATA
Enthusiasm
WOW!
SO FEW BUGS!
Time


TOTALLY SCENTIFIC DATA
Enthusiasm
CAN HACK WITHOUT FEAR
ON NEOW FEATURE
Time


TOTALLY SCENTIFIC DATA
I DON'T ALWAYS WANT
TO WORK ON MY PROJECT
BUT WHEN I DO
I FEEL LIKE A BOSS
Enthusiasm
Time

What's unique about Rust?

- The (dreaded) borrow checker
- Great for having this low overhead
- But for high-level applications...
  - "What if we defaulted to using `Arc<RefCell<T>>` everywhere?"
  - Rust as OCaml with different syntax?
- Is Rust be ok-ish for high-level *despite* the borrow checker?
- Or could the borrow checker be actually, you know, good for that too?

The dreaded borrow checker

- A reference can be
  - shared
  - mutable
  - pick one
- Memory safety!
- Prevents data races!

The dreaded borrow checker

- A reference can be
  - shared
  - mutable
  - pick one
- Memory safety!
- Prevents data races!
- Readable code?

Example

```
print a
foo()
bar.foo()
print a # Can be different!
```

- Hard to reason locally
- Have to know the details of every function/method
- Spaghetti!

Example in Rust

```rust
let a = some_stuff();
println!("{:?}", a);
foo();
bar.foo();
println!("{:?}", a); // Is the same!
```

Example in Rust (bis)

```rust
let a = some_stuff();
println!("{:?}", a);
foo(&mut a);
{
    let bar = &mut a;
    bar.foo();
}
println!("{:?}", a); // Has changed
```

The dreaded borrow checker

- Ensures that
  - If you're using something, no-one interferes with it
    - Unless it's explicit
  - If you modify something you haven't meddled where you shouldn't have
- Makes it easier to reason about code
- Helps with safety and data races
- But also with correctness, maintenance, ...

Personal again

- Two kind of borrow checker issues
  - Easy to fix, but annoying ones
    - `vec.push(vec.len())`
  - Hard ones where you have to step back and think about your design
- Having to refactor is painful and time-consuming
- But it avoid later problems that can be more painful and time consuming
- If you really have to get around it (`Cell`, `Mutex`, not event talking about `unsafe`), you *really* should know what you are doing

Alternative: functional programming

- No mix between sharing and mutability
- No mutability at all!
- Simple to reason about the code
- Except... you have to get used to it
- Have to rewire your brain somehow

Monads

Position in the landscape of PL

- Imperative programming
  - Alias and mutation $\rightarrow$ openbar
  - Garbage collection mitigats the problem but doesn't solve it
- Functional programming
  - No mutation at all
- Rust
  - Controlled mutability
- Not just interesting for systems programming
- Provides another alternative

In summary

- Short-term VS long-term
- Rust can be hard at first, but it makes things easier later on
  - Bugs
  - Maintenance
  - Accepting contributions
- Problem: you might not see the time you gain
  - Since you don't have to spend days finding the bugs
- The compiler is grumpy for you so you don't have to be

Rust

- For beginner programmers?
  - Probably not ideal
  - You have to learn quite a few things
  - I hope I'm wrong :)
- For great programmers?
  - Not needed
- For average programmers?
  - Who might not have the best design at first try
  - Who might forget some things sometimes
  - Who might sometimes be lazy
  - Basically, mere mortals
  - Makes thing "easier" in the long run

What can be improved?

Language

- Borrow checking is necessary but can be refined
  - Non-lexical lifetimes
  - Nested method calls
- Some $*$ could be removed
  - Match
  - Auto-deref for generic functions/methods `&*(?! )`
- Object-oriented
  - Delegate implementation
  - Specialisation

Ecosystem

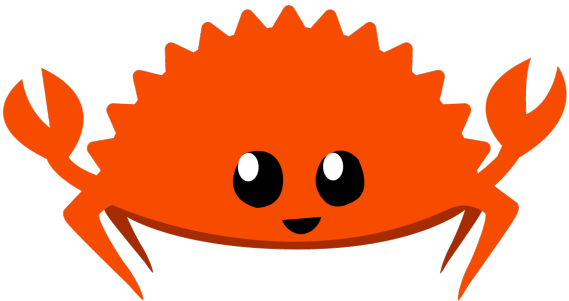
- YMMV, but younger than Java's, C#, Python's, Ruby's, ...

Personnal list

- Internationalisation
- Lack of guidelines for writing applications
- Some kind of crates.io-like platform where you can publish the code of an app and it
  - Builds it
  - Cross-compiles it
  - Packages it to .deb/rpm/whatever

Community

- Welcoming community
- Increasing Rust's reach
- Rustbridge



Thanks!

- Questions?
- Twitter: @lise__henry (or @crowdagger if you can bear french and flood)
- Github: lise-henry