The Usability of Advanced Type Systems: Rust as a Case Study

Kasra Ferdowsi

04/27/2022

Overview

1. History
2. Rust
3. Usability
4. Future Work

Overview

1. History
    1.1. Linear Types
    1.2. Region-based Memory Management
    1.3. Ownership Types
2. Rust
3. Usability
4. Future Work

1.1 Linear Types

- "Use *Exactly* Once" [25]:

```
copy :: a → (a, a)
copy a = (a, a)
```

```
drop :: a → ()
drop a = ()
```

1.1 Linear Types

- "Use *Exactly* Once" [25]:

```
copy :: a → (a, a)        drop :: a → ()
copy a = (a, a)           drop a = ()
```

- Memory management:

```
append :: List → List → List
append = λxs → …
```

1.1 Linear Types

- "Use *Exactly* Once" [25]:

```
copy :: a → (a, a)        drop :: a → ()
copy a = (a, a)           drop a = ()
```

- Memory management:

```
append :: ¡List -o ¡List -o ¡List
append = ¡λxs → …
```

1.1 Linear Types

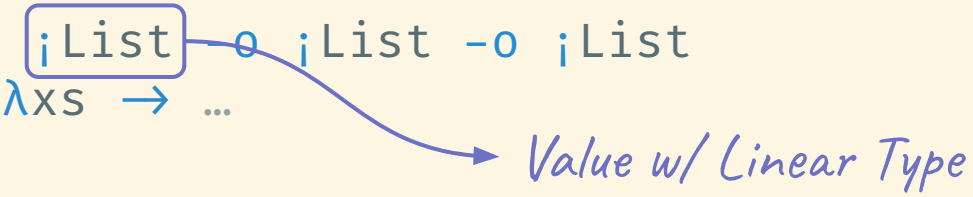
- "Use *Exactly* Once" [25]:

```
copy :: a → (a, a)
copy a = (a, a)
```

```
drop :: a → ()
drop a = ()
```

- Memory management:

```
append :: ¡List -o ¡List -o ¡List
append = λxs → …
```



1.1 Linear Types

- "Use *Exactly* Once" [25]:

```
copy :: a → (a, a)          drop :: a → ()
copy a = (a, a)             drop a = ()
```

- Memory management:

```
append :: ¡List -o ¡List -o ¡List
append = λxs → …
```

Function w/ Linear Param

1.2 Region-based Memory Management

- Tofte and Talpin 1997 [24]

1.2 Region-based Memory Management

- ~~Tofte and Talpin 1997 [24]~~

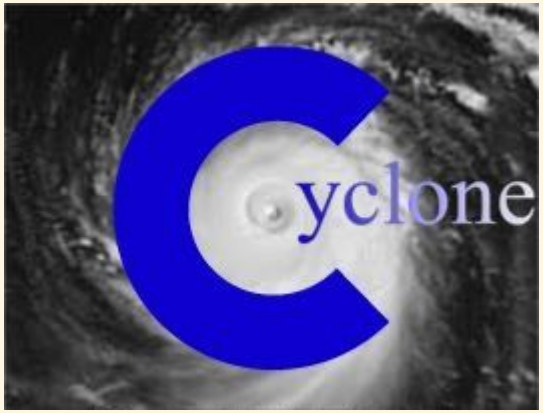
  Cyclone! [16]

1.2 Region-based Memory Management

- ~~Tofte and Talpin 1997 [24]~~

  Cyclone! [16]

"A Safe Dialect of C"



1.2 Region-based Memory Management

- Region Types [16]

```
region h {
  int* x = rmalloc(h, sizeof(int));
  int? y = rnew(h) { 1, 2, 3 };
  char? z = rprintf(h, "hello");
}
```

1.2 Region-based Memory Management

- Region Types [16]

```
region h {
  int* x = rmalloc(h, sizeof(int));
  int? y = rnew(h) { 1, 2, 3 };
  char? z = rprintf(h, "hello");
}
```

maps to a dynamic growable region of memory

1.2 Region-based Memory Management

- Region Types [16]

```
region h {
  int* x = rmalloc(h, sizeof(int));
  int? y = rnew(h) { 1, 2, 3 };
  char? z = rprintf(h, "hello");
}
```

Pointer types are annotated with the name of the region 'h

1.2 Region-based Memory Management

- Region Types [16]

```
region h {
  int*`h x = rmalloc(h, sizeof(int));
  int?`h y = rnew(h) { 1, 2, 3 };
  char?`h z = rprintf(h, "hello");
}
```

Pointer types are annotated with the name of the region `h

1.3 Ownership Types [6]

- Statically enforcing Encapsulation

1.3 Ownership Types [6]

- Statically enforcing Encapsulation

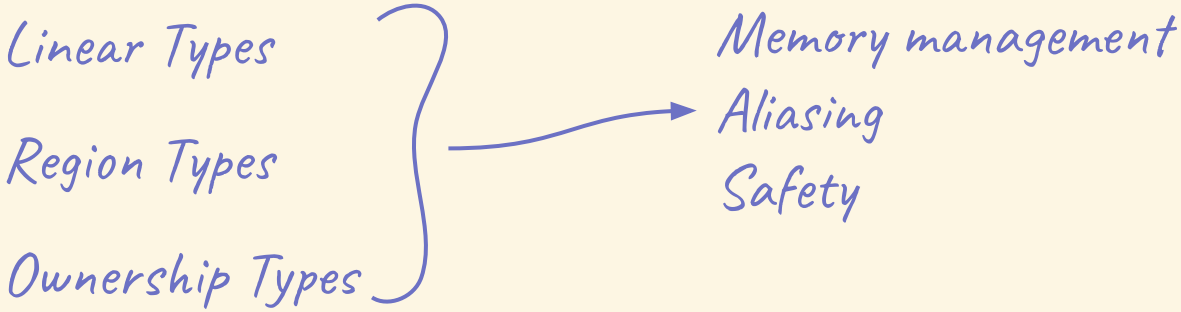
1 History

Linear Types

Region Types

Ownership Types

1 History


Linear Types
Region Types
Ownership Types
Memory management
Aliasing
Safety

Overview

1. History
    1.1. Linear Types
    1.2. Region-based Memory Management
    1.3. Ownership Types
2. Rust
3. Usability
4. Future Work

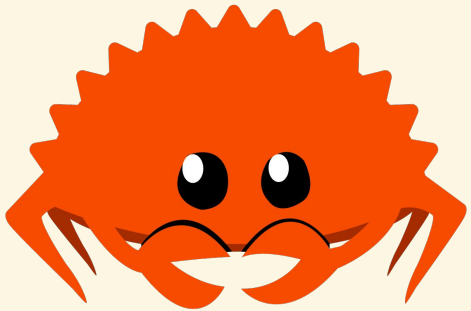
Overview

1. History
2. Rust
    2.1. Ownership
    2.2. Lifetimes
3. Usability
4. Future Work

2 Rust

- Started in 2010 @ Mozilla Research
- Mascot is *Ferris*
- Its ~~cultists~~ are know as "Rustaceans"

community



2 Rust

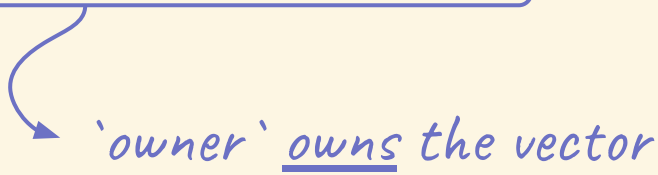
- Ownership & Borrowing

```
fn main() {
  let owner: Vec<i32> = vec![1, 2, 3];
}
```

2 Rust

- Ownership & Borrowing

```rust
fn main() {
    let owner: Vec<i32> = vec![1, 2, 3];
}
```



2 Rust

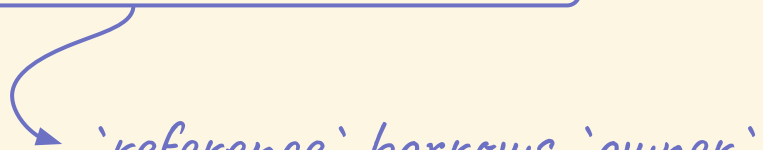
- Ownership & Borrowing

```
fn main() {
  let owner: Vec<i32> = vec![1, 2, 3];
  let reference: &Vec<i32> = &owner;
}
```

2 Rust

- Ownership & Borrowing

```
fn main() {
  let owner: Vec<i32> = vec![1, 2, 3];
  let reference: &Vec<i32> = &owner;
}
```



2 Rust

- Mutability

```
fn main() {
  let owner: Vec<i32> = vec![1, 2, 3];
  owner.push(4);

  let reference: &Vec<i32> = &owner;
}
```

2 Rust

- Mutability

```
fn main() {
  let owner: Vec<i32> = vec![1, 2, 3];
  owner.push(4);
  ^^^^^^^^^^^^ cannot borrow as mutable
  let reference: &Vec<i32> = &owner;
}
```

2 Rust

- Mutability

```
fn main() {
  let mut owner: Vec<i32> = vec![1, 2, 3];
  owner.push(4);

  let reference: &Vec<i32> = &owner;
}
```

2 Rust

- Mutability

```
fn main() {
  let mut owner: Vec<i32> = vec![1, 2, 3];
  owner.push(4);

  let reference: &Vec<i32> = &owner;
}
```

2 Rust

- Mutability

```
fn main() {
  let mut owner: Vec<i32> = vec![1, 2, 3];
  owner.push(4);

  let reference: &mut Vec<i32> = &mut owner;
  reference.push(5);
}
```

2 Rust

- Mutability

```
fn main() {
  let mut owner: Vec<i32> = vec![1, 2, 3];
  owner.push(4);

  let reference: &mut Vec<i32> = &mut owner;
  reference.push(5);
}
```

2.1 Ownership

- The three rules of Ownership [9]

2.1 Ownership

- The three rules of Ownership [9]
- Checked by the Borrow-Checker

2.1 Ownership

1. All values have exactly one owner.

2.1 Ownership

1. All values have exactly one owner.

```
fn main() {
  let v = vec![1, 2, 3];
  let v2 = v;
  print(v);
}
```

2.1 Ownership

1. All values have exactly one owner.

```
fn main() {
  let v = vec![1, 2, 3];
  let v2 = v;                    value moved here
  print(v);                      value used here after move
}
```

2.1 Ownership

2. A reference to a value cannot outlive the owner.

2.1 Ownership

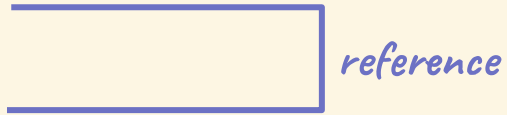
2. A reference to a value cannot outlive the owner.

```
fn main() {
  let v = vec![1, 2, 3];
  let x = &v[0];
  let v2 = v;
  let y = x + 1;
}
```

2.1 Ownership

2. A *reference* to a value cannot *outlive* the owner.

```
fn main() {
  let v = vec![1, 2, 3];
  let x = &v[0];
  let v2 = v;
  let y = x + 1;
}
```



2.1 Ownership

2. A *reference* to a value cannot *outlive* the owner.

```
fn main() {
  let v = vec![1, 2, 3];
  let x = &v[0];
  let v2 = v;
  let y = x + 1;
}
```

2.1 Ownership

2. A *reference* to a value cannot *outlive* the owner.

```
fn main() {
  let v = vec![1, 2, 3];
  let x = &v[0];              borrow out of `v` occurs here
  let v2 = v;                 move out of `v` occurs here
  let y = x + 1;              borrow later used here
}
```

2.1 Ownership

3. A value can have one *mutable* ref, or many *immutable* refs.

2.1 Ownership

3. A value can have one *mutable* ref, or many *immutable* refs.

```
fn main() {
  let mut v = vec![1, 2, 3];
  let x = &v[0];
  v.push(4);
  let y = x + 1;
}
```

2.1 Ownership

3. A value can have one *mutable* ref, or many *immutable* refs.

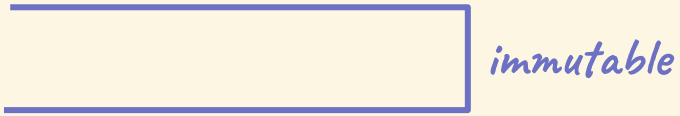
```
fn main() {
  let mut v = vec![1, 2, 3];
  let x = &v[0];
  v.push(4);
  let y = x + 1;
}
```

```
fn push(&mut self, value: i32) { /* ... */ }
```

2.1 Ownership

3. A value can have one *mutable* ref, or many *immutable* refs.

```
fn main() {
  let mut v = vec![1, 2, 3];
  let x = &v[0];
  v.push(4);
  let y = x + 1;
}
```




```
fn push(&mut self, value: i32) { /* ... */ }
```

2.1 Ownership

3. A value can have one *mutable* ref, or many *immutable* refs.

```
fn main() {
  let mut v = vec![1, 2, 3];
  let x = &v[0];
  v.push(4);
  let y = x + 1;
}
```


```
fn push(&mut self, value: i32) { /* ... */ }
```

2.1 Ownership

3. A value can have one *mutable* ref, or many *immutable* refs.

```
fn main() {
  let mut v = vec![1, 2, 3];
  let x = &v[0];            immutable borrow occurs here
  v.push(4);                mutable borrow occurs here
  let y = x + 1;            immutable borrow later used here
}
```

2.1 Ownership

1. All values have exactly one owner.
2. A reference to a value cannot outlive the owner.
3. A value can have:
   - one mutable reference, or
   - many immutable references.

2.1 Ownership

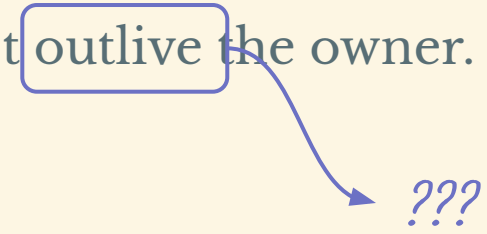
1. All values have exactly one owner.
2. A reference to a value cannot outlive the owner.
3. A value can have:
   - one mutable reference, or
   - many immutable references.

2.1 Ownership

1. All values have exactly one owner.
2. A reference to a value cannot outlive the owner.
3. A value can have:
   - one mutable reference, or
   - many immutable references.

2.1 Ownership

1. All values have exactly one owner.
2. A reference to a value cannot outlive the owner.
3. A value can have:
   - one mutable reference, or
   - many immutable references.



2.1 Ownership

1. All values have exactly one owner.
2. A reference to a value cannot outlive the owner.
3. A value can have:
   - one mutable reference or
   - many immutable references.

Lifetimes

2.2 Lifetimes

Lifetimes [17]:

```
fn main() {
  let mut v = vec![1, 2, 3];
  let x = &v[0];
  v.push(4);
  let y = x + 1;
}
```

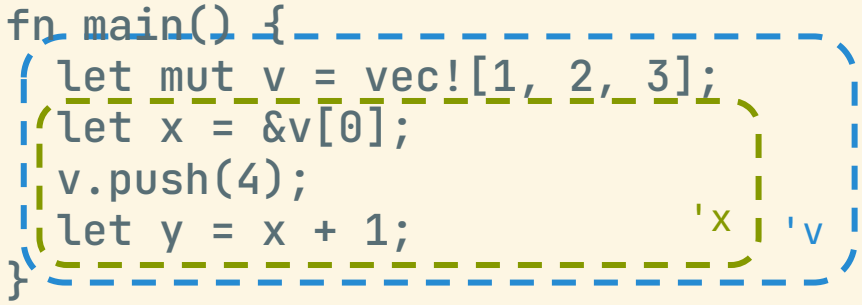
2.2 Lifetimes

Lifetimes [17]:

```
fn main() {
    let mut v = vec![1, 2, 3];
    let x = &v[0];
    v.push(4);
    let y = x + 1;
}
```
'v

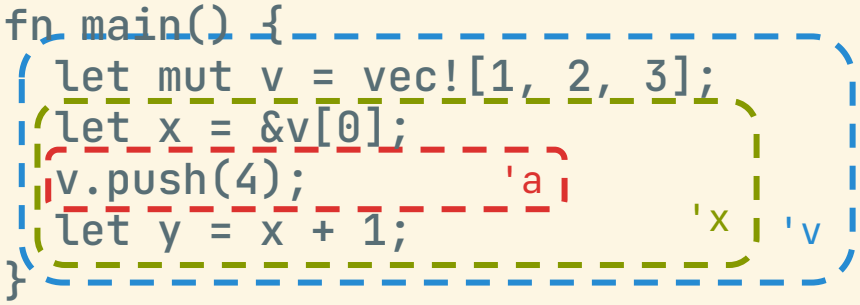
2.2 Lifetimes

Lifetimes [17]:



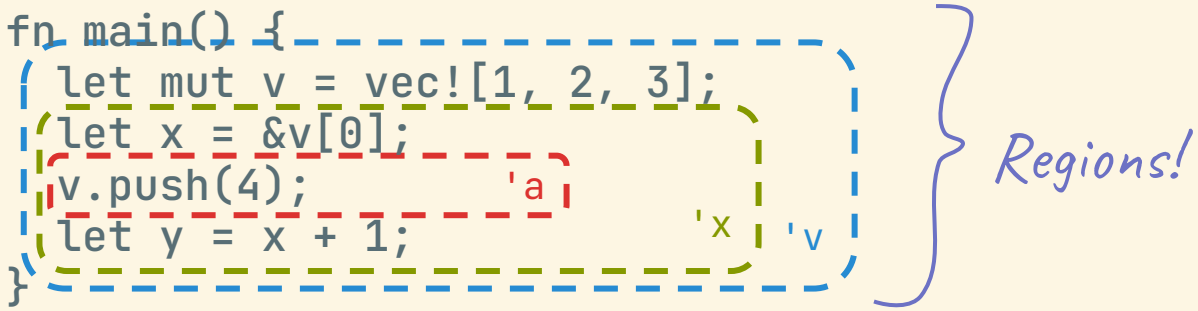
2.2 Lifetimes

Lifetimes [17]:



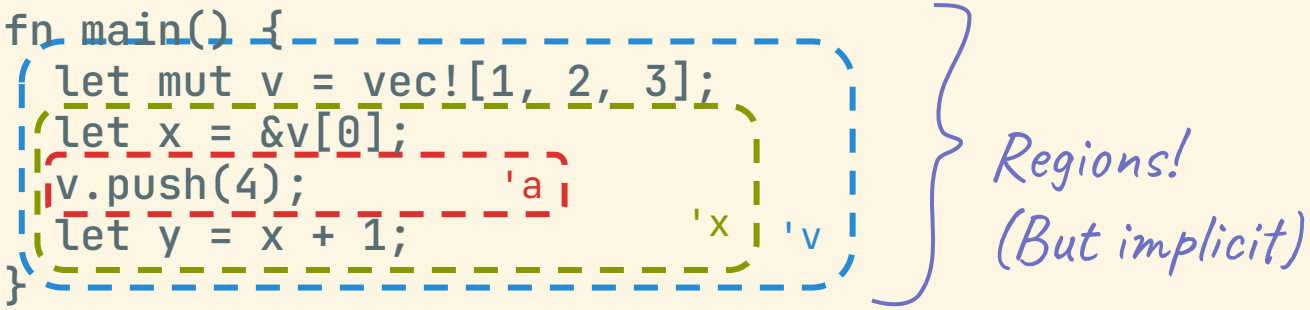
2.2 Lifetimes

Lifetimes [17]:



2.2 Lifetimes

Lifetimes [17]:



2 Rust

Quick break!

Any questions so far?

Overview

1. History
2. Rust
    a. Ownership
    b. Lifetimes
3. Usability
4. Future Work

Overview

1. History
2. Rust
3. Usability
4. Future Work

3.1 Is Rust Difficult?

"Learning Rust Ownership is like navigating a maze where the walls are made of asbestos and frustration, and the maze has no exits, and every time you hit a dead end you get an aneurysm and die." [8]

Overview

1. History
2. Rust
3. Usability
    3.1. *Is* Rust's Ownership difficult to learn & use?
4. Future Work

Overview

1. History
2. Rust
3. Usability
    3.1. *Is* Rust's Ownership difficult to learn & use?
    3.2. *Why* is Rust's Ownership difficult?
4. Future Work

Overview

1. History
2. Rust
3. Usability
    - 3.1. *Is* Rust's Ownership difficult to learn & use?
    - 3.2. *Why* is Rust's Ownership difficult?
    - 3.3. Why do people use Rust anyway?
4. Future Work

3.1 Is Rust Difficult?

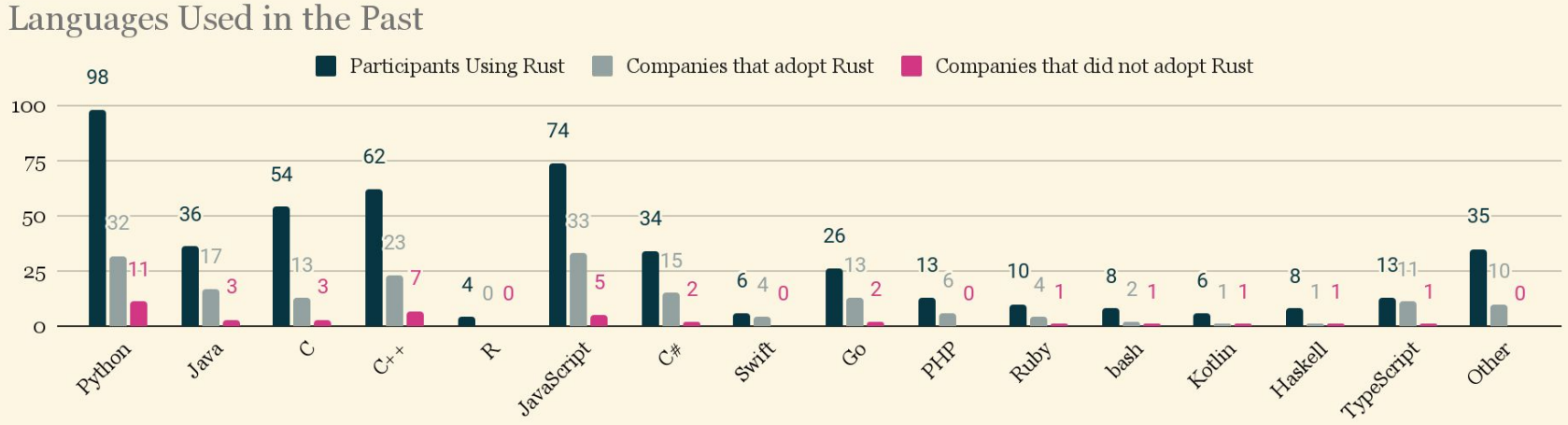
- Zeng and Crichton 2019 [26]

  "The complexity of the borrow checker was the second most frequently mentioned complaint"

3.1 Is Rust Difficult?

- Fulton and friends 2021 [12]

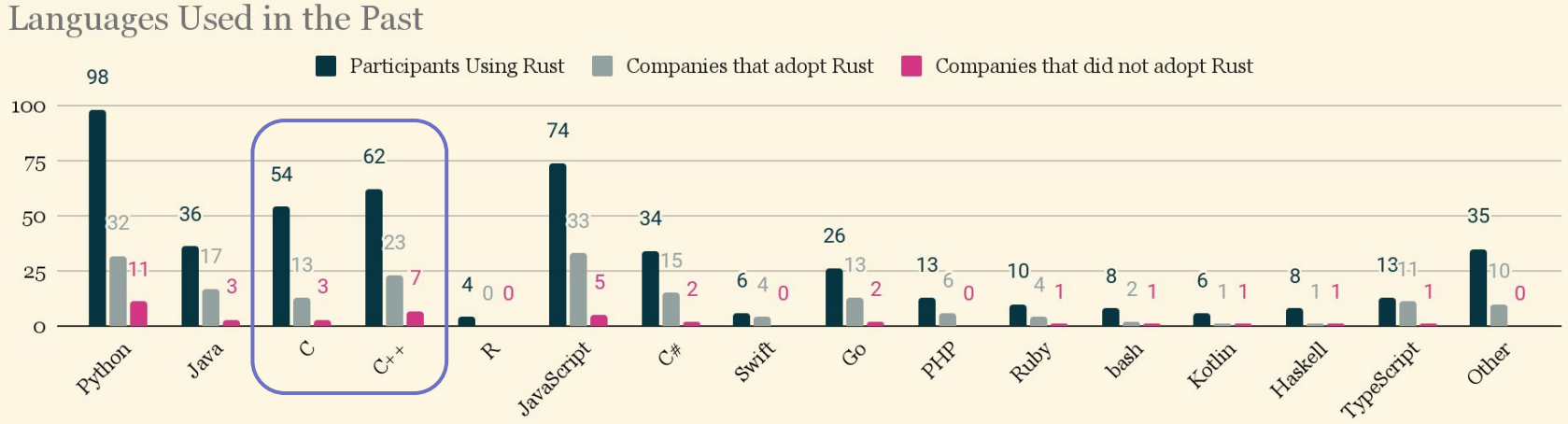
  "A near-vertical learning curve"



3.1 Is Rust Difficult?

- Fulton and friends 2021 [12]

  "A near-vertical learning curve"

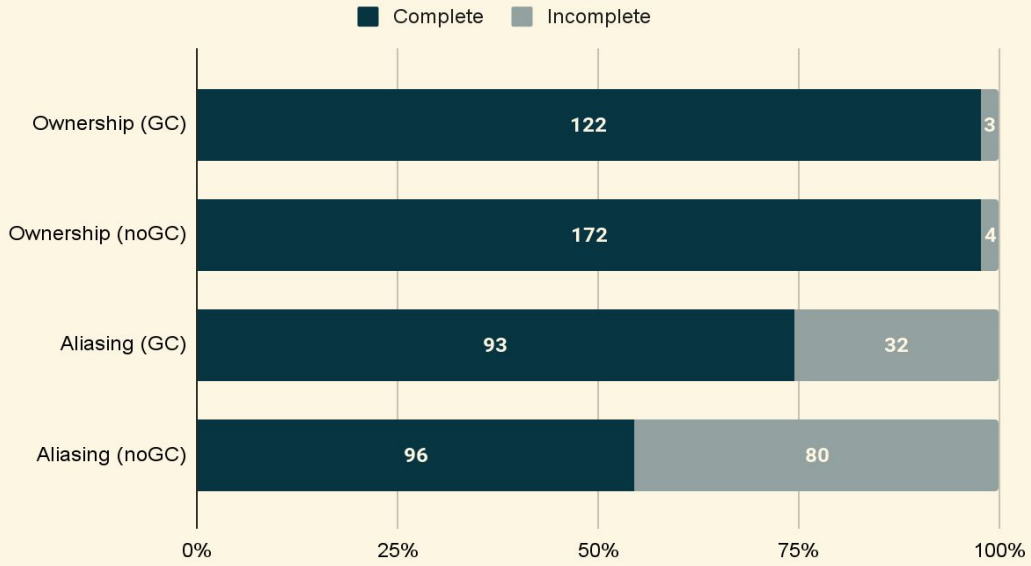


3.1 Is Rust Difficult?

- The Bronze Garbage Collector [8]

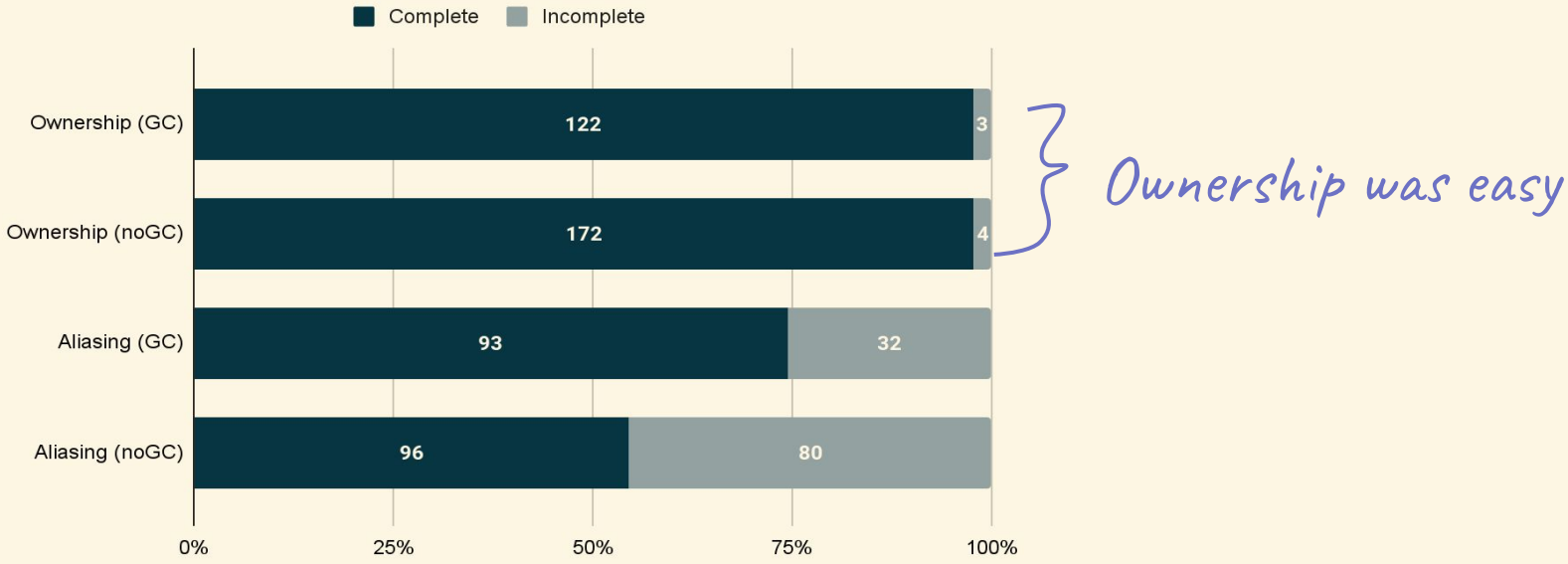
3.1 Is Rust Difficult?

- The Bronze Garbage Collector [8]



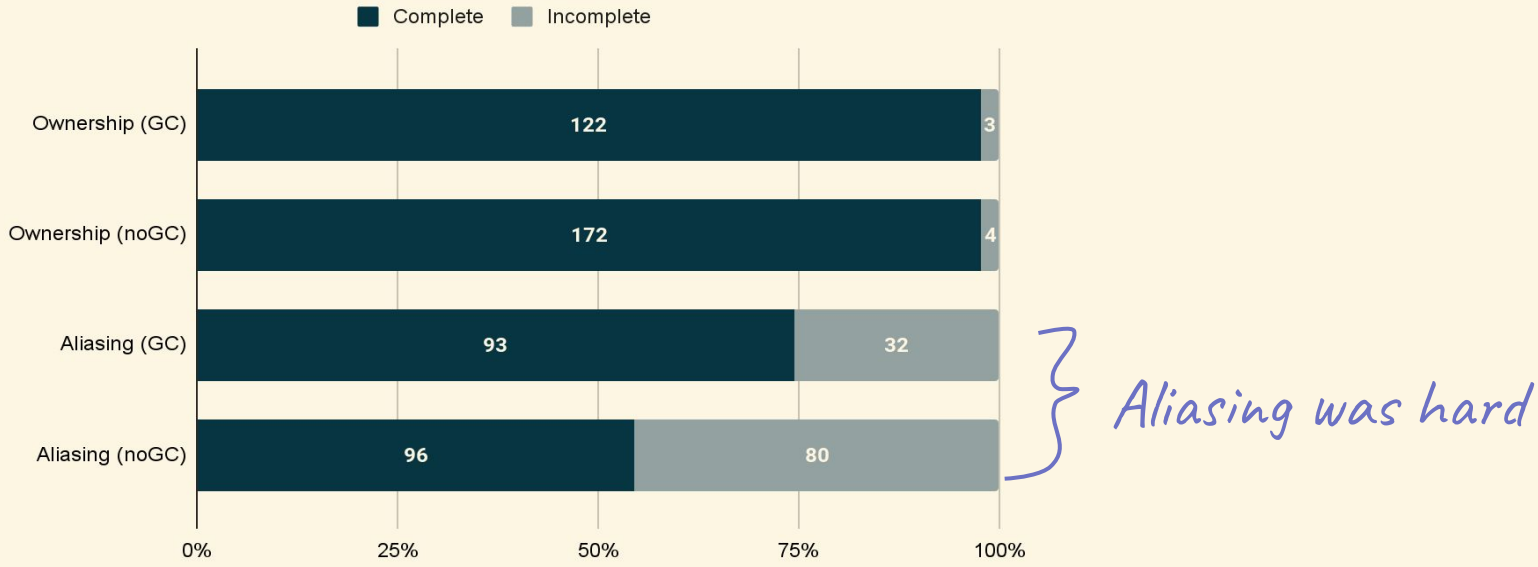
3.1 Is Rust Difficult?

- The Bronze Garbage Collector [8]



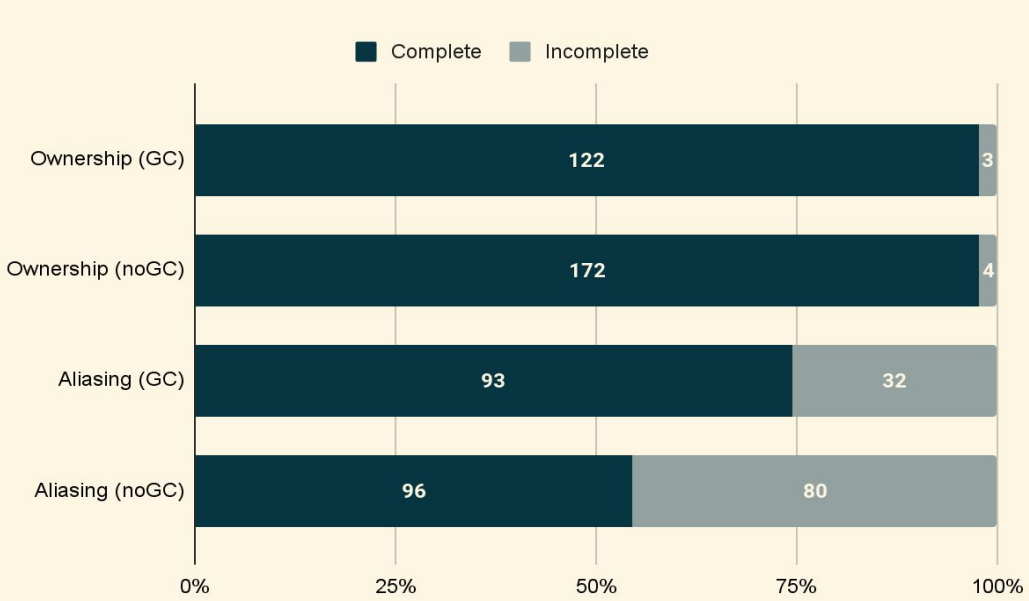
3.1 Is Rust Difficult?

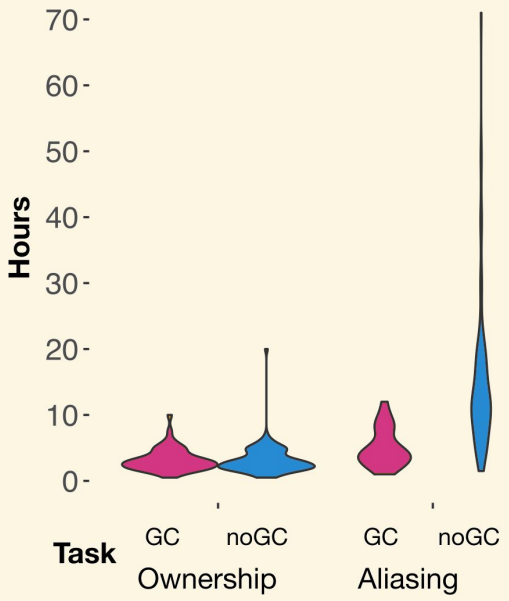
- The Bronze Garbage Collector [8]



3.1 Is Rust Difficult?

- The Bronze Garbage Collector [8]





3.1 Is Rust Difficult?

- What about experienced Rust developers?

3.1 Is Rust Difficult?

- What about experienced Rust developers?
  - Qin & friends [19]: "Lacking good understanding of Rust's **lifetime** rules"

3.1 Is Rust Difficult?

- What about experienced Rust developers?
  - Qin & friends [19]: "Lacking good understanding of Rust's **lifetime** rules"
  - The Rust Survey 2020 [20]: "**lifetimes**"

3.1 Is Rust Difficult?

- What about experienced Rust developers?
  - Qin & friends [19]: "Lacking good understanding of Rust's **lifetime** rules"
  - The Rust Survey 2020 [20]: "**lifetimes**"
  - Zhu & friends [27]: "complex **lifetime** computations"

3.1 Is Rust Difficult?

3.1 Is Rust Difficult?

Yes!

Lifetimes are difficult even for experienced Rustaceans.

3.2 Why Is Rust Difficult?

3.2 Why Is Rust Difficult?

Walls of text coming...

Sorry!

3.2 Why Is Rust Difficult?

- Change of Paradigm
  - "interference" & "mindshift" [21]

3.2 Why Is Rust Difficult?

- Change of Paradigm
  - "interference" & "mindshift" [21]
  - Common memory access patterns are unsafe [26]

3.2 Why Is Rust Difficult?

- Change of Paradigm
  - "interference" & "mindshift" [21]
  - Common memory access patterns are unsafe [26]
  - "having to redesign code that you know is safe, but the compiler doesn't" [12]

3.2 Why Is Rust Difficult?

- Change of Paradigm
  - "interference" & "mindshift" [21]
  - Common memory access patterns are unsafe [26]
  - "having to redesign code that you know is safe, but the compiler doesn't" [12]
  - "a significant part of the benefit of GC in Rust programs is the architectural simplification", and
  - "design was a significant contributor to the difference in performance between non-Bronze and Bronze participants." [8]

3.2 Why Is Rust Difficult?

- Change of Paradigm
  - "interference" & "mindshift" [21]
  - Common memory access patterns are unsafe [26]
  - "having to redesign code that you know is safe, but the compiler doesn't" [12]
  - "a significant part of the benefit of GC in Rust programs is the architectural simplification", and
  - "design was a significant contributor to the difference in performance between non-Bronze and Bronze participants." [8]
  - Rust's borrow-checker "forces a programmer to think differently" [21]

3.2 Why Is Rust Difficult?

- Change of Paradigm
  - "interference" & "mindshift" [21]
  - Common memory access patterns are unsafe [26]
  - "having to redesign code that you know is safe, but the compiler doesn't" [12]
  - "a significant part of the benefit of GC in Rust programs is the architectural simplification", and
  - "design was a significant contributor to the difference in performance between non-Bronze and Bronze participants." [8]
  - Rust's borrow-checker "forces a programmer to think differently" [21]
  - *Not just because it's a low-level language!*

3.2 Why Is Rust Difficult?

- Error Messages

3.2 Why Is Rust Difficult?

- Error Messages

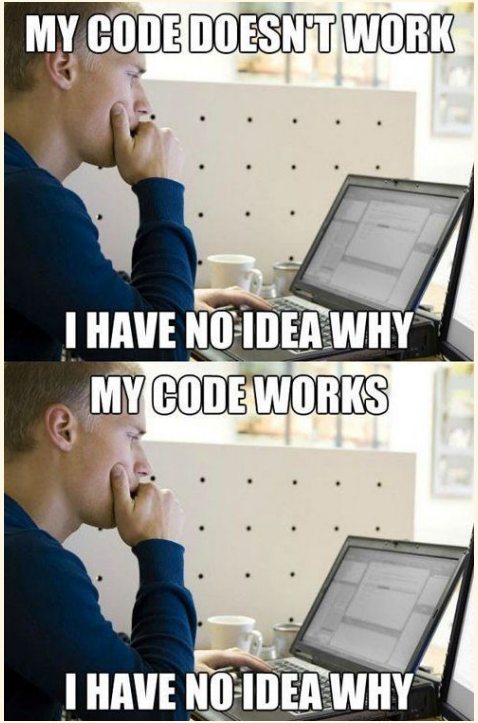
```
error[E0384]: cannot assign twice to immutable variable `x`
  ⟶ blah.rs:7:5
   |
6  |  let x = 1;
   |      -
   |      |
   |      first assignment to `x`
   |      help: consider making this binding mutable: `mut x`
7  |  x += 1;
   |  ^^^^^^ cannot assign twice to immutable variable

error: aborting due to previous error

For more information about this error, try `rustc --explain E0384`.
```

3.2 Why Is Rust Difficult?

- Error Messages & Design Feedback [8]

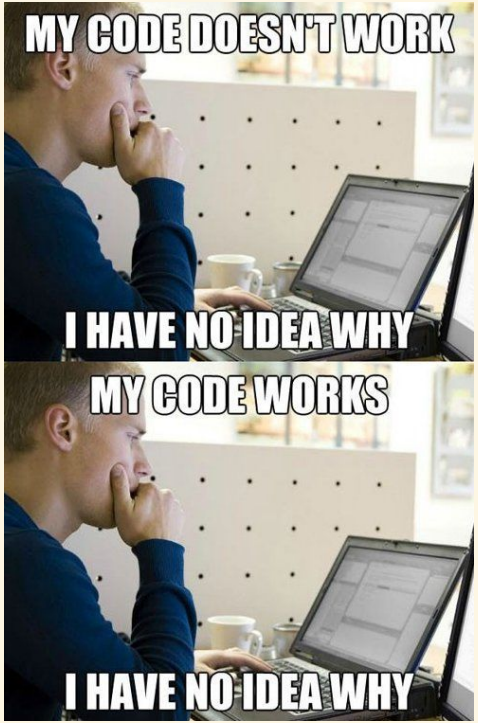
3.2 Why Is Rust Difficult?

- Error Messages & Design Feedback [8]



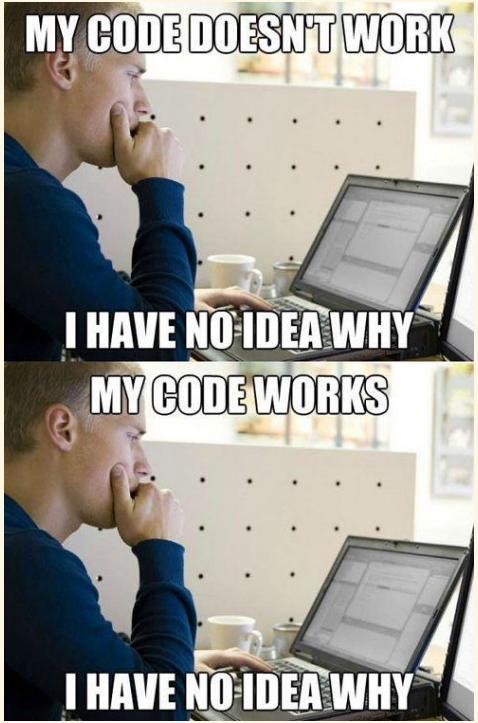
3.2 Why Is Rust Difficult?

- Error Messages & Design Feedback [8]
  - "Getting weird errors I still don't understand but just fixed by listening to the compiler"



3.2 Why Is Rust Difficult?

- Error Messages & Design Feedback [8]
  - "Getting weird errors I still don't understand but just fixed by listening to the compiler"
  - "Error messages were cyclical with things like `remove &` then after removing `try adding &`"



3.2 Why Is Rust Difficult?

- Error Messages & Context [27]

3.2 Why Is Rust Difficult?

- Error Messages & Context [27]
  - Most (59 / 110) contained all necessary information,
  - 9 failed to explain how a safety rule works on a particular code construct,
  - 32 were missing the key steps in computing a lifetime or a borrow relationship,
  - 10 failed to explain the relationship between two lifetime annotations.

3.2 Why Is Rust Difficult?

- The Curse of Incompleteness [9]

3.2 Why Is Rust Difficult?

- The Curse of Incompleteness [9]

> “I can teach the three rules [of Ownership] in a single lecture to a room of undergrads. But the vagaries of the borrow checker still trip me up every time I use Rust!”

3.2 Why Is Rust Difficult?

- The Curse of Incompleteness [9]

```rust
let mut v = vec![1, 2];
let x = &mut v[0];
let y = &mut v[1];
*y += *x;
```

```rust
let mut v = vec![1, 2];
let mut iter = v.iter_mut();
let x = iter.next().unwrap();
let y = iter.next().unwrap();
*y += *x;
```

3.2 Why Is Rust Difficult?

- The Curse of Incompleteness [9]

```
let mut v = vec![1, 2];
let x = &mut v[0];
let y = &mut v[1];
*y += *x;
```

```
let mut v = vec![1, 2];
let mut iter = v.iter_mut();
let x = iter.next().unwrap();
let y = iter.next().unwrap();
*y += *x;
```

3.2 Why Is Rust Difficult?

- The Curse of Incompleteness [9]

```
let mut v = vec![1, 2];
v.insert(0, v[0]);
v.get_mut(v[0]);
```

3.2 Why Is Rust Difficult?

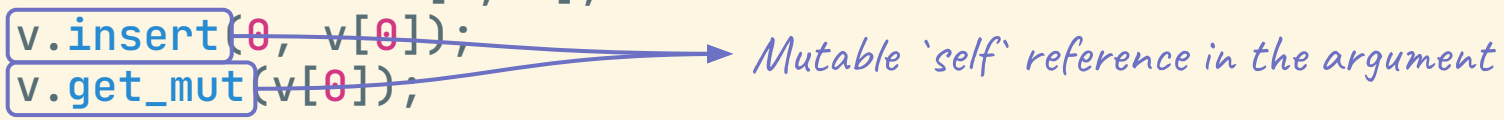
- The Curse of Incompleteness [9]

```
let mut v = vec![1, 2];
v.insert(0, v[0]);
v.get_mut(v[0]);
```

3.2 Why Is Rust Difficult?

- The Curse of Incompleteness [9]

```
let mut v = vec![1, 2];
v.insert(0, v[0]);
v.get_mut(v[0]);
```



3.2 Why Is Rust Difficult?

- The Curse of Incompleteness [9]

```
let mut v = vec![1, 2];
v.insert(0, v[0]); ✓
v.get_mut(v[0]);   ✗
```

3.2 Why Is Rust Difficult?

3.2 Why Is Rust Difficult?

1. Needs a change of paradigm
2. Error messages are missing design feedback and necessary context
3. Incompleteness makes building an accurate mental model difficult

3.3 Why do people use Rust anyway?

3.3 Why do people use Rust anyway?

Read the report!

(Sorry! ☹ *)*

3.3 Why do people use Rust anyway?

1. They don't
2. Safety is the most important
3. Tooling is great
4. `unsafe` code is used a lot, and for good reason

Overview

1. History
2. Rust
3. Usability
4. Future Work

4 Future Work

Better Error Messages [2]

4 Future Work

Better Error Messages [2]

- Programmers *do* read error messages

4 Future Work

Better Error Messages [2]

- Programmers *do* read error messages
- Rust follows many of the guidelines, but can do better

4 Future Work

Better Error Messages [2]

- Programmers *do* read error messages
- Rust follows many of the guidelines, but can do better

Provide Context [4,27]

4 Future Work

Better Error Messages [2]

- Programmers *do* read error messages
- Rust follows many of the guidelines, but can do better

Provide Context [4,27] → Reduce Cognitive Load

4 Future Work

Better Error Messages [2]

- Programmers *do* read error messages
- Rust follows many of the guidelines, but can do better

Provide Context [4,27]

Reduce Cognitive Load

Allow Dynamic Interaction

4 Future Work

How do Rust programmers write code?

4 Future Work

How do Rust programmers write code?

- Grounded Theory [5]

4 Future Work

How do Rust programmers write code?

- Grounded Theory [5]
- Used in Software Engineering Research [23]

4 Future Work

How do Rust programmers write code?

- Grounded Theory [5]

Mis Used in Software Engineering Research [23]

4 Future Work

How do Rust programmers write code?

- Grounded Theory [5]

Mis Used in Software Engineering Research [23]

- How Statically-Typed Functional Programmers Write Code [18]

4 Future Work

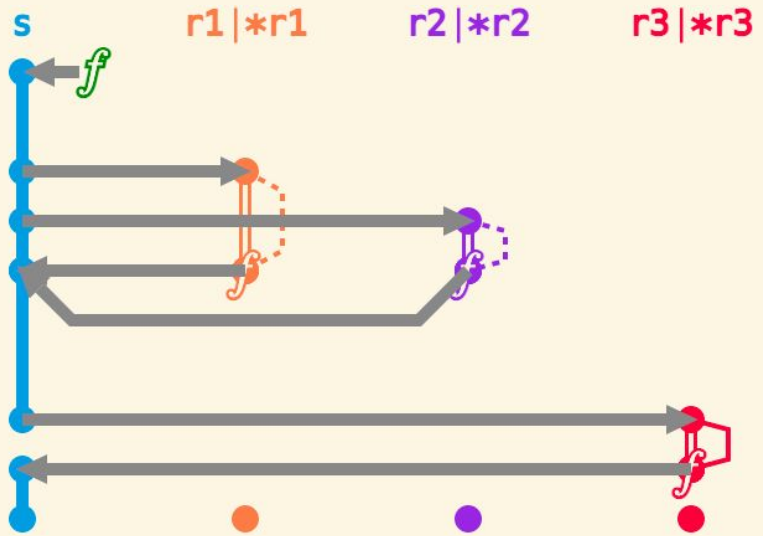
Program Visualization Tools

4 Future Work

Program Visualization Tools

- RustViz [13]

```
1  fn main(){
2      let mut s = String::from("hello");
3
4      let r1 = &s;
5      let r2 = &s;
6      assert!(compare_strings(r1, r2));
7
8      let r3 = &mut s;
9      clear_string(r3);
10 }
```

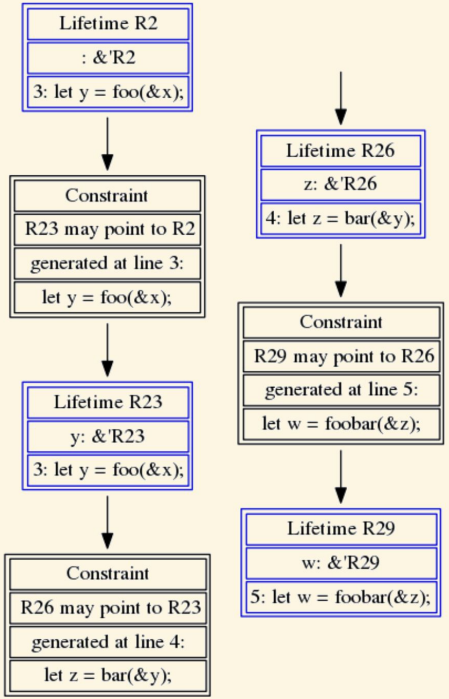


4 Future Work

Program Visualization Tools

- Rust Life Assistant [4]

```
fn main() {
      let mut x = 4;
      let y = foo(&x);
      let z = bar(&y);
      let w = foobar(&z);
      // ...
      x = 5;
      take(w);
}

fn foo<T>(p: T) → T { p }
fn bar<T>(p: T) → T { p }
fn foobar<T>(p: T) → T { p }
fn take<T>(p: T) → T { unimplemented!() }
```



4 Future Work

Program Visualization Tools

- Existing tools focus on *runtime* values [22]
- Rust needs a tool for *static* type constraints

Overview

1. History
    a. Linear Types
    b. Region-based Memory Management
    c. Ownership Types
2. Rust
    a. Ownership
    b. Lifetimes
3. Usability
    a. *Is* Rust's Ownership difficult?
    b. *Why* is Rust's Ownership difficult?
    c. Why do people use Rust anyway?
4. Future Work
    a. Better Error Messages
    b. How Do Rust programmers Write Code?
    c. Program Visualization Tools

Bibliography

[1] Aldrich, J. et al. 2002. Alias Annotations for Program Understanding. *Proceedings of the 17th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (New York, NY, USA, 2002), 311–330.

[2] Becker, B.A. et al. 2019. Compiler Error Messages Considered Unhelpful: The Landscape of Text-Based Programming Error Message Research. *Proceedings of the Working Group Reports on Innovation and Technology in Computer Science Education* (New York, NY, USA, Dec. 2019), 177–210.

[3] Bernardy, J.-P. et al. 2017. Linear Haskell: practical linearity in a higher-order polymorphic language. *Proceedings of the ACM on Programming Languages*. 2, POPL (Dec. 2017), 5:1-5:29. DOI:https://doi.org/10.1145/3158093.

[4] Blaser, D. 2019. *Simple Explanation of Complex Lifetime Errors in Rust*. ETH Z ̈urich.

[5] Charmaz, K. 2006. *Constructing Grounded Theory: A Practical Guide Through Qualitative Analysis*. SAGE Publishing Inc.

[6] Clarke, D. et al. 2013. Ownership Types: A Survey. *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*. D. Clarke et al., eds. Springer. 15–58.

[7] Clarke, D.G. et al. 1998. Ownership Types for Flexible Alias Protection. *Proceedings of the 13th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (New York, NY, USA, 1998), 48–64.

[8] Coblenz, M. et al. 2021. Does the Bronze Garbage Collector Make Rust Easier to Use? A Controlled Experiment. *arXiv:2110.01098 [cs]*. (Oct. 2021).

[9] Crichton, W. 2021. The Usability of Ownership. *arXiv:2011.06171 [cs]*. (Sep. 2021).

[10] Dominik, D. 2018. *Visualization of Lifetime Constraints in Rust*. ETH Z ̈urich.

Bibliography (cont.)

[11] Fahndrich, M. and DeLine, R. 2002. Adoption and Focus: Practical Linear Types for Imperative Programming. *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation* (New York, NY, USA, 2002), 13–24.

[12] Fulton, K.R. et al. 2021. Benefits and drawbacks of adopting a secure programming language: rust as a case study. *Seventeenth Symposium on Usable Privacy and Security (SOUPS 2021)* (2021), 597–616.

[13] Gongming et al. 2020. RustViz: Interactively Visualizing Ownership and Borrowing. *arXiv:2011.09012 [cs]*. (Nov. 2020).

[14] Grossman, D. et al. 2005. Cyclone: A type-safe dialect of C. *C/C++ Users Journal*. 23, 1 (2005), 112–139.

[15] Grossman, D. et al. 2002. Region-based memory management in cyclone. *ACM SIGPLAN Notices*. 37, 5 (May 2002), 282–293. DOI:https://doi.org/10.1145/543552.512563.

[16] Jim, T. et al. 2002. Cyclone: A safe dialect of C. *Proc. of the 2002 USENIX Annual Technical Conference* (Jan. 2002), 275–288.

[17] Jung, R. et al. 2019. Stacked borrows: an aliasing model for Rust. *Proceedings of the ACM on Programming Languages*. 4, POPL (Dec. 2019), 41:1-41:32. DOI:https://doi.org/10.1145/3371109.

[18] Lubin, J. and Chasins, S.E. 2021. How statically-typed functional programmers write code. *Proceedings of the ACM on Programming Languages*. 5, OOPSLA (Oct. 2021), 155:1-155:30. DOI:https://doi.org/10.1145/3485532.

[19] Qin, B. et al. 2020. Understanding memory and thread safety practices and issues in real-world Rust programs. *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, Jun. 2020), 763–779.

[20] Rust Survey 2020 Results: 2020. *https://blog.rust-lang.org/2020/12/16/rust-survey-2020.html*. Accessed: 2022-01-31.

Bibliography (cont.)

[21] Shrestha, N. et al. 2020. Here We Go Again: Why is It Difficult for Developers to Learn Another Programming Language? *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (New York, NY, USA, 2020), 691–701.

[22] Sorva, J. et al. 2013. A Review of Generic Program Visualization Systems for Introductory Programming Education. *ACM Transactions on Computing Education*. 13, 4 (Nov. 2013), 1–64. DOI:https://doi.org/10.1145/2490822.

[23] Stol, K.-J. et al. 2016. Grounded theory in software engineering research: a critical review and guidelines. *Proceedings of the 38th International Conference on Software Engineering* (New York, NY, USA, May 2016), 120–131.

[24] Tofte, M. and Talpin, J.-P. 1997. Region-Based Memory Management. *Information and Computation*. 132, 2 (Feb. 1997), 109–176. DOI:https://doi.org/10.1006/inco.1996.2613.

[25] Wadler, P. 1990. Linear Types can Change the World! *Programming Concepts and Methods* (1990).

[26] Zeng, A. and Crichton, W. 2019. Identifying Barriers to Adoption for Rust through Online Discourse. *arXiv:1901.01001 [cs]*. (Jan. 2019).

[27] Zhu, S. et al. 2022. Learning and Programming Challenges of Rust: A Mixed-Methods Study. (2022), 13.

1.1 Linear Types

- Elsewhere in Linear Types:
  a. Vault [11] to Linear Haskell [3]
  b. Region [23] and Ownership types [7]

1.2 Region-based Memory Management

- ~~Tofte and Talpin 1997~~

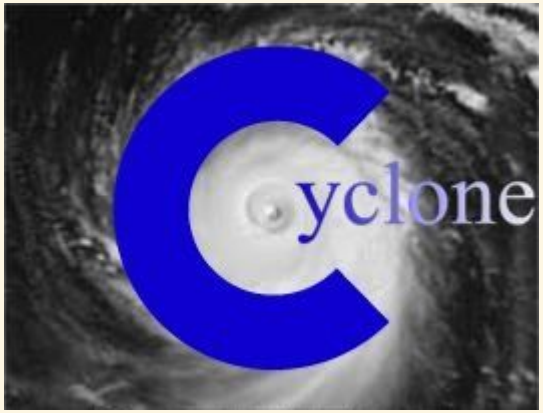
  Cyclone! [15]

Tagged Unions NULL checks

"A Safe Dialect of C"

Existential Types

RBMM

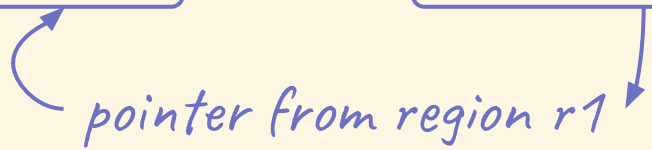


1.2 Region-based Memory Management

- Region Types [15]

```
region h {
  int* x = rmalloc(h, sizeof(int));
  int? y = rnew(h) { 1, 2, 3 };
  char? z = rprintf(h, "hello");
}
```

```
char?`r rprintf(region_t<`r> r1, const char? fmt, /*…*/);
```



1.3 Ownership Types [6]

- Statically enforcing Encapsulation

```
class Rectangle {
  private Point upperLeft;
  private Point lowerRight;

  public Rectangle(Point ul, Point lr) {
    upperLeft = ul;
    lowerRight = lr;
  }

  public Point getUpperLeft() {
    return new Point(upperLeft.x, upperLeft.y);
  }
}
```

1.3 Ownership Types [6]

- AliasJava [1]

```
class Rectangle {
  private owned Point upperLeft;
  private owned Point lowerRight;

  public Rectangle(unique Point ul, unique Point lr) {
    upperLeft = ul;
    lowerRight = lr;
  }

  public unique Point getUpperLeft() {
    return new Point(upperLeft.x, upperLeft.y);
  }
}
```

1.4 Shared Ideas

- Benefits
  a. Memory management
  b. Safety
  c. Comprehension
- Expressiveness
  a. Linear Types: non-linear types + read-only references
  b. RBMM: garbage-collected heap region
  c. Ownership types: external uniqueness

2.3 `unsafe`

- "Escape Hatch", but not really:
    a. Dereference a raw pointer
    b. Call an unsafe function or method
    c. Access or modify a mutable static variable
    d. Implement an unsafe trait
    e. Access fields of unions

2.3 unsafe

```
unsafe fn two_mutable_refs(x: &mut i32) → (&mut i32, &mut i32) {
    (&mut *(x as *mut i32), &mut *(x as *mut i32))
}

fn main() {
    let mut x = 1;
    let (c1, c2) = unsafe { two_mutable_refs(&mut x) };
    *c1 += 1;
    *c2 += 1;
    println!("{}", x);
}
```

2.1 Locking Bugs

```
fn do_request() {
    // client: Arc<RwLock<Inner>>
    match connect(client.read().unwrap().m) {
        Ok(_) ⇒ {
            let mut inner = client.write().unwrap();
            inner.m = mbrs;
        }
        Err(_) ⇒ {}
    }
}
```

2.1 Locking Bugs

```
fn do_request() {
    // client: Arc<RwLock<Inner>>
    let result = connect(client.read().unwrap().m);
    match result {
        Ok(_) ⇒ {
            let mut inner = client.write().unwrap();
            inner.m = mbrs;
        }
        Err(_) ⇒ {}
    }
}
```

2.1 Locking Bugs

```
fn do_request() {
    // client: Arc<RwLock<Inner>>
-   match connect(client.read().unwrap().m) {
+   let result = connect(client.read().unwrap().m);
+   match result {
        Ok(_) ⇒ {
            let mut inner = client.write().unwrap();
            inner.m = mbrs;
        }
        Err(_) ⇒ {}
    }
}
```