OWNERSHIP AND ~~LIFETIME~~ DRIVEN SYNTHESIZER FOR AUTOMATIC C TO RUST TRANSLATION

Meng Wang & Hanliang Zhang

University of Bristol

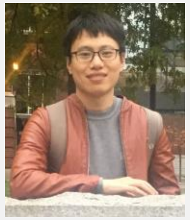
A FEW WORDS ABOUT ME

- Reader (Assoc. Prof.) at University of Bristol
- Head of Bristol Programming Languages Group https://bristolpl.github.io/
- Research interest — Correctness of Programs
  - Programming Language Design
  - Functional Programming
  - Testing
  - Program Synthesis

WORK ON RUST

- First encounter in 2012 (less than 2 years of its inception)
- Seriously working on it since 2021 (with a UK government funded project)
- Bristol is becoming a major center of Rust research in the UK

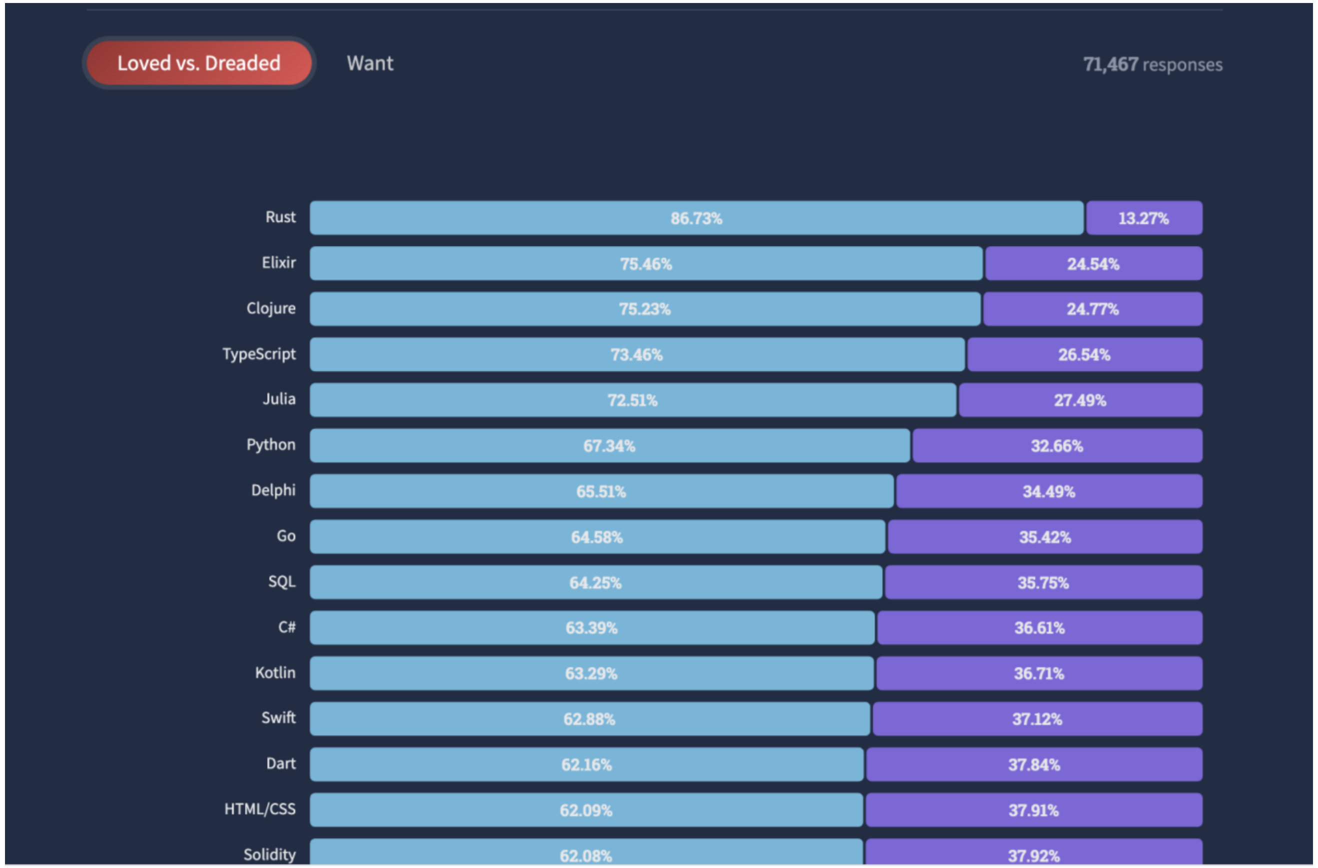


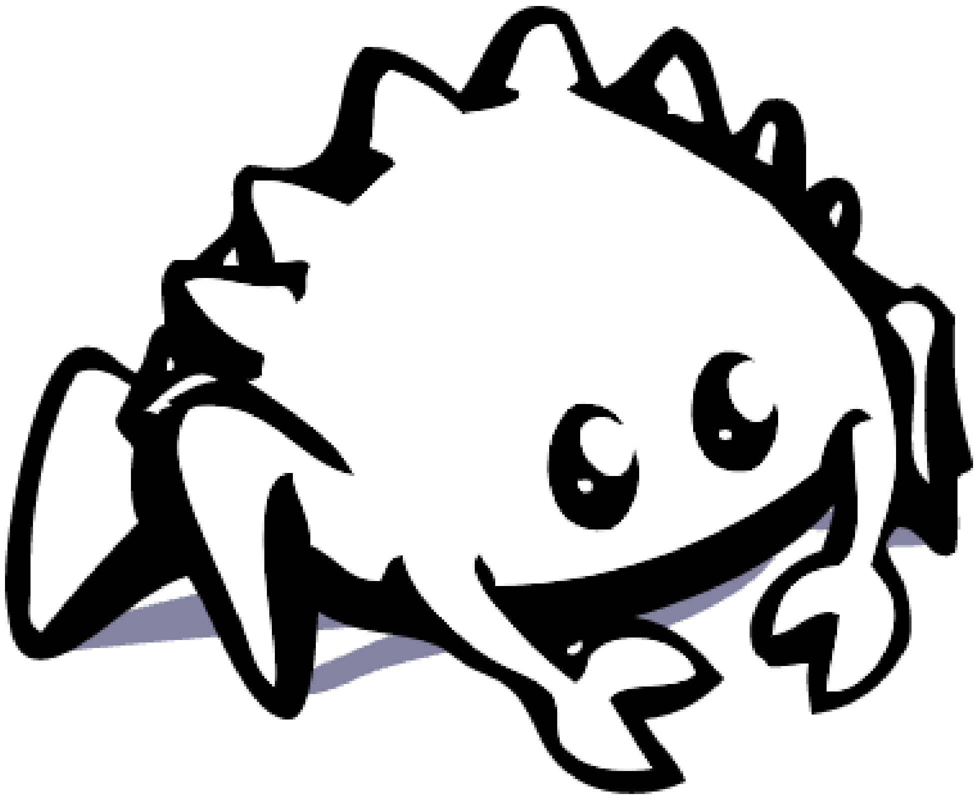




- Actively collaborating with Huawei since 2022

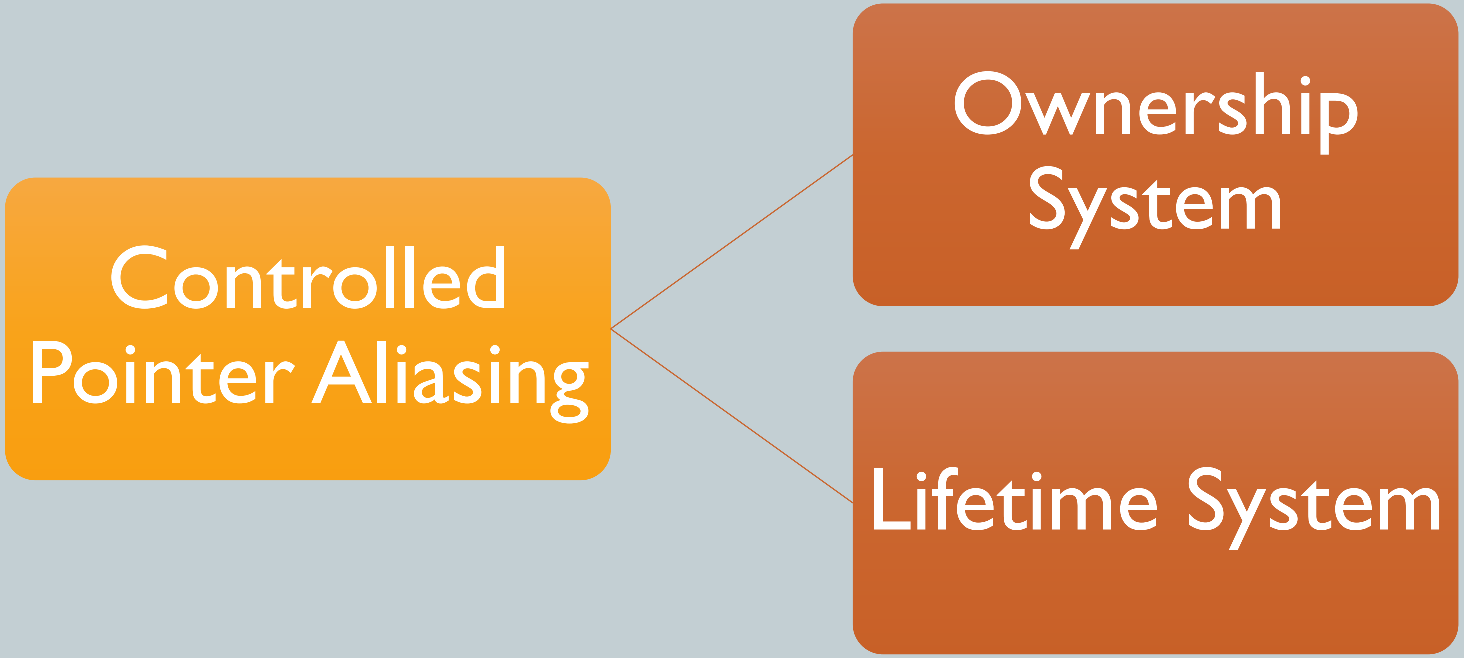
RUST: WHAT'S THE FUSS?

- Multi-paradigm
- C-like syntax
- Memory safety
- Efficient





RUST'S MEMORY SAFETY



OWNERSHIP

Rust's ownership rules:

- Each value in Rust has a variable that's called its owners
- There can only be one owner at a time
- When the owner goes out of scope, the value will be dropped

OWNERSHIP

Uniqueness of owned pointers (no alias)

```
fn f() {
    let p = Box::new(0); // p is the owner
    let q = p; // ownership is transferred to q
    // p cannot be used later on
    let _ = q; // q is consumed
}
```

Avoiding DOUBLE-FREE

BORROWING

Manually passing ownership around can be tedious in programming:

```
fn foo(v1: Vec<i32>, v2: Vec<i32>) -> (Vec<i32>, Vec<i32>, i32) {
    // Do stuff with `v1` and `v2`.

    // Hand back ownership, and the result of our function.
    (v1, v2, 42)
}

let v1 = vec![1, 2, 3];
let v2 = vec![1, 2, 3];

let (v1, v2, answer) = foo(v1, v2);
```

- Borrowing allows ownership to be temporarily transferred and automatically returned after use
- An address to data like the address taking operation in C and C++
- Borrowing comes with two kinds of permission
  - &T a read-only reference
  - &mut T a mutable reference

LIFETIME

- **Exclusive-Or Principle**: At any given time, you can have *either* one mutable reference *or* any number of immutable references
- **Lifetime** is another mechanism used by the Rust compiler to ensure this principle.

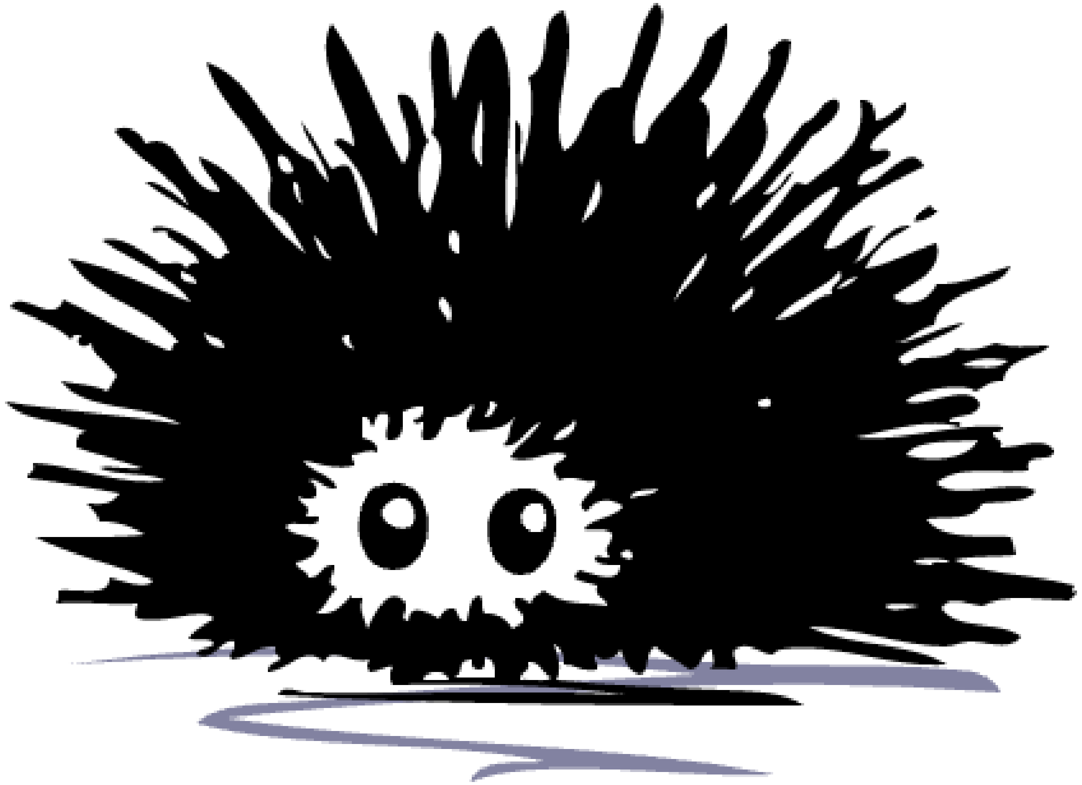
LIFETIME

```
fn exclusive_or() {
    let mut local = 0

    let p = &local;
    let q = &mut local; // compilation error!

    read(p);
    mutate(q);
}
```

Mutable, unique reference



UNSAFE RUINS (OR ENABLES?) EVERYTHING

```
fn f() {
    unsafe {
        let p: *mut i32 = malloc(4) as *mut _;
        let q: *mut i32 = p;
        free(q as *mut ());
    }
}
```

- Unrestricted Pointer Aliasing…
- It is now programmers' responsibility to ensure memory safety

POINTERS IN C VS RUST

Pointers in C: raw pointers

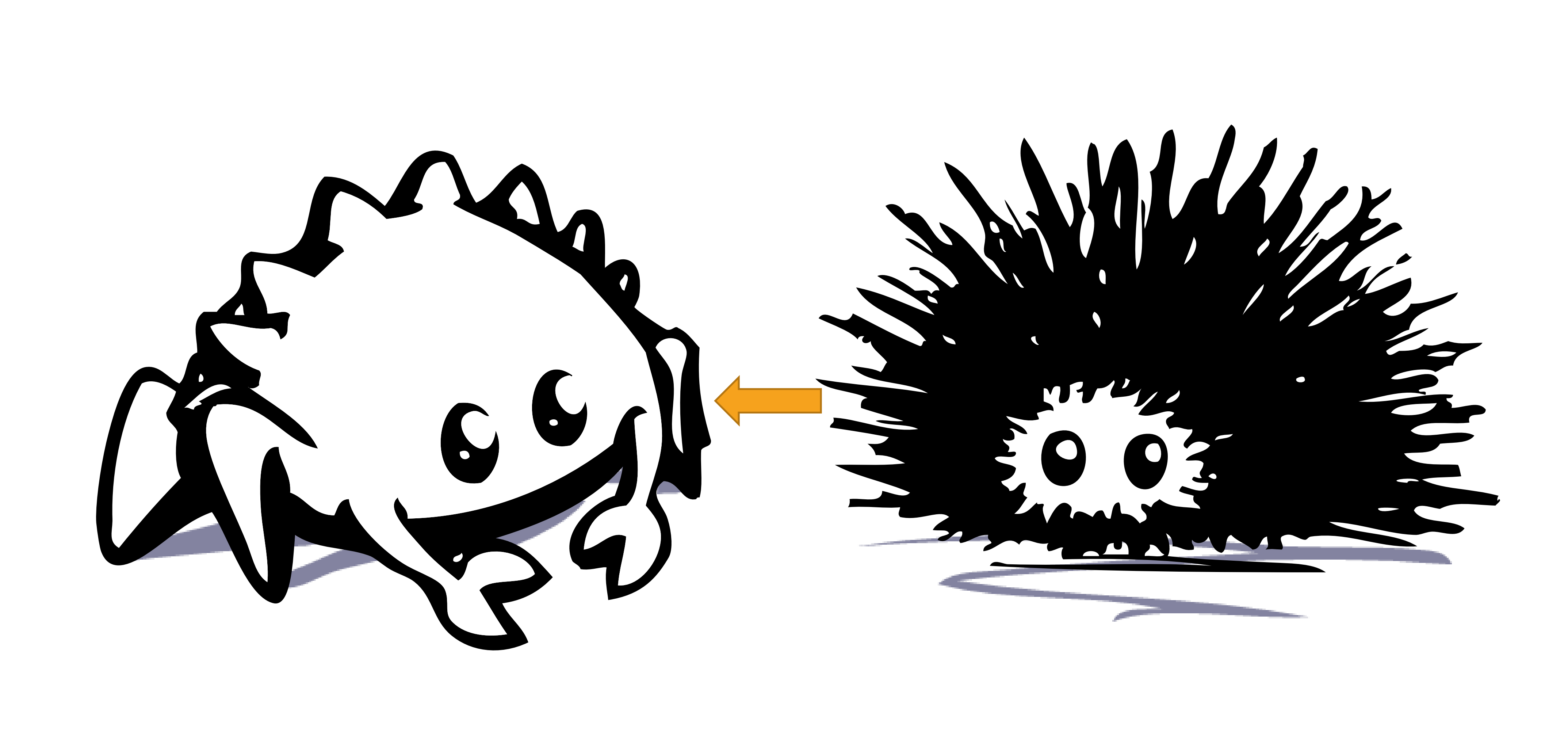
Pointers in Rust:

1. Raw Pointers
2. Reference (&mut T, &T), which
   is temporary borrow of some value
3. Smart Pointers (for example Box<T>),
   that support proper memory
   management.

EXISTING TECHS ON TRANSLATING C TO RUST

- C2Rust
  - Industrial strength Syntax-Directed Basic C to Rust Translator
  - 0% unsafe code ratio
- CRustS (Huawei)
  - Preprocessing steps that remove unnecessary unsafe, fix build errors, etc.
  - Limited success at expression-level unsafe ratio
- Laertes [Emre et el. OOPSLA21]
  - Extended rewrite steps that rely solely on the compiler error msg.
  - Ad hoc approach that does not make use of the core ownership concept of safe Rust

```
fn f() {
    unsafe {
        let p: *mut i32 = malloc(4) as *mut _;
        let q: *mut i32 = p;
        free(q as *mut ());
    }
}
```

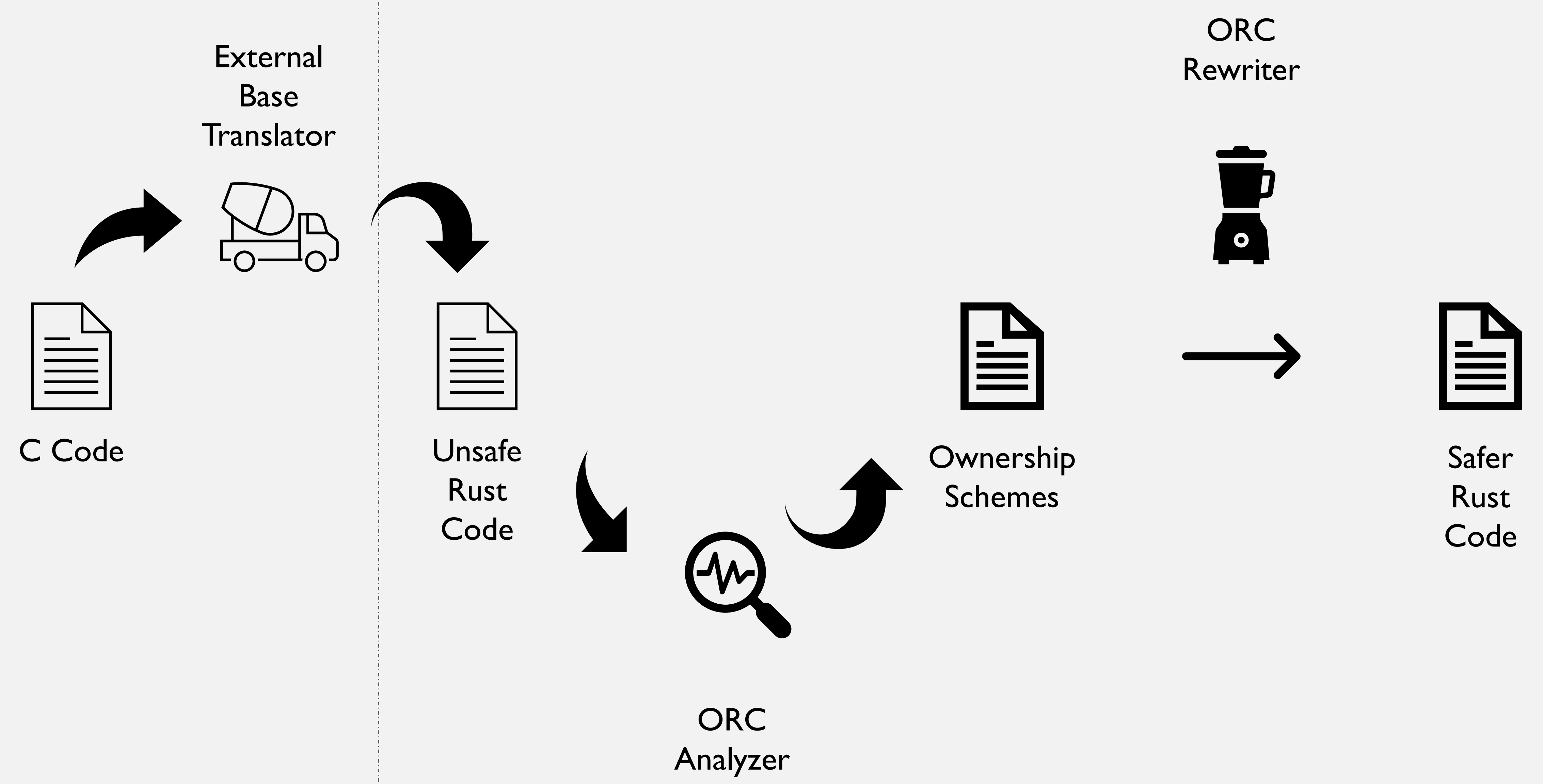


ORC

Orc focuses on the ownership model of safe Rust. It tries to discover an underlying 'ownership scheme' for an unsafe Rust program (typically translated from C program). W.r.t. this scheme, Orc re-types pointers and rewrite their usages into safer ones.

'Ownership Scheme': Is a pointer in charge of allocating, releasing some computational resource at a specific program point? If so, is the role of allocation and deallocation unique to this pointer?

In other words, does this pointer conceptually own some computational resources?


C Code
External Base Translator
Unsafe Rust Code
ORC Analyzer
Ownership Schemes
ORC Rewriter
Safer Rust Code

```c
void f() {
  int* p = malloc(sizeof(int));

  int* q = p;
  free(q);
}
```

C2Rust

Is it possible to find a proper ownership scheme for this program?

```rust
fn f() {
    unsafe {
        let p: *mut i32 = malloc(4) as *mut _;
        let q: *mut i32 = p;
        free(q as *mut ());
    }
}
```

A possible ownership transfer

Does p own an integer?

Yes!

```
fn f() {
    unsafe {
        let p: *mut i32 = malloc(4) as *mut _; // p allocates an integer
        let q: *mut i32 = p;                    // ???
        free(q as *mut ());                     // q frees an integer
    }
}
```

How about q?

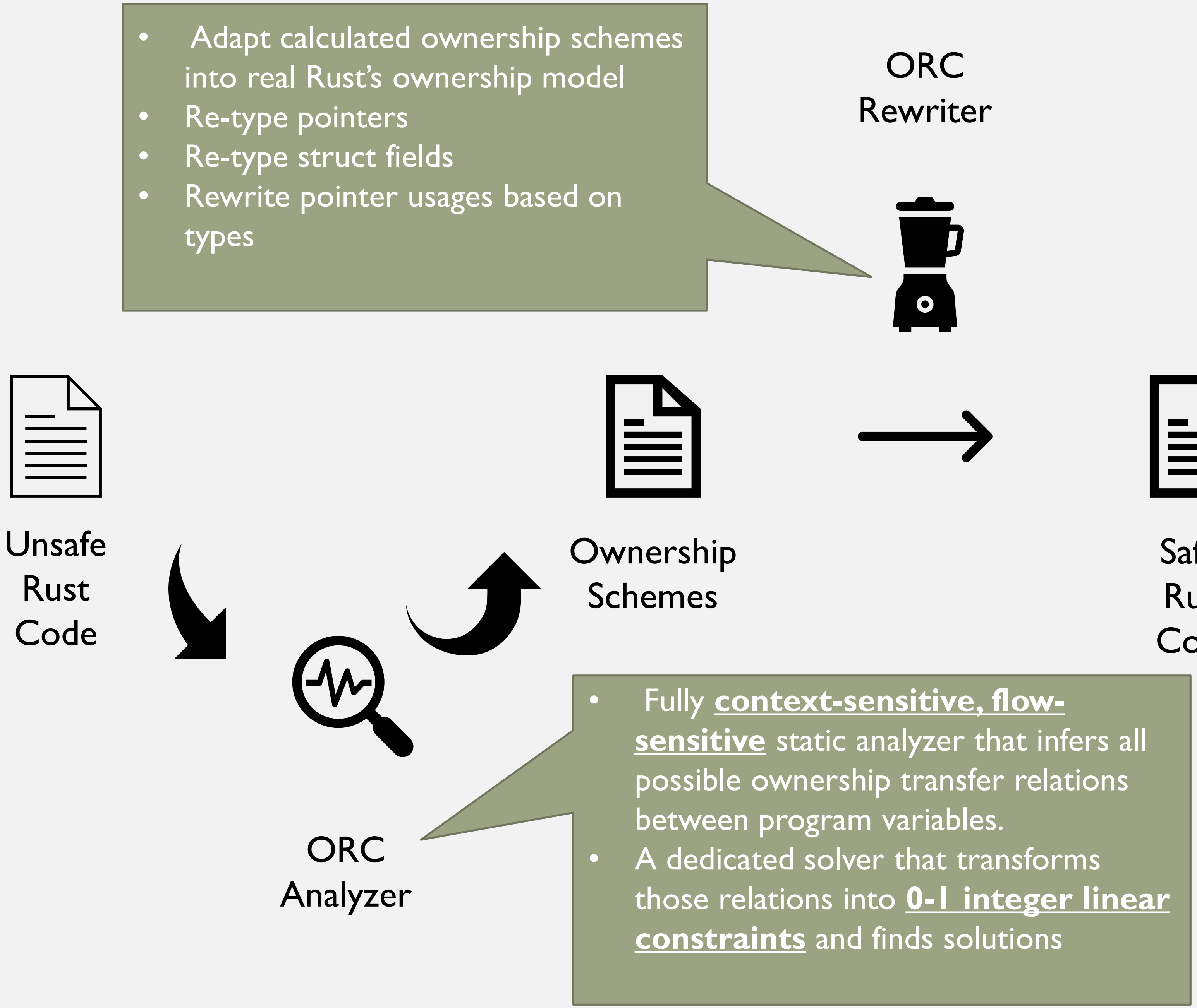
Yes!

1. On line 1, p allocates an integer. At this program point, p conceptually owns this integer
2. One line 2, the ownership of this integer is transferred from p to q (this guarantees uniqueness)
3. On line 3, q is responsible for releasing resources, therefore q should have ownership

```rust
fn f() {
    unsafe {
        let p: *mut i32 = malloc(4) as *mut _;
        let q: *mut i32 = p;
        free(q as *mut ());
    }
}
```

The ownership scheme found by ORC analyzer is then adapted to real Rust's ownership model, by re-typing owning pointers to Box pointers.

```rust
fn f_safe() {
    let p = Box::new(i32::default());
    let q = p; // ownership transfer
    let _ = q;
}
```



Adapt calculated ownership schemes into real Rust's ownership model
Re-type pointers
Re-type struct fields
Rewrite pointer usages based on types
ORC Rewriter
Unsafe Rust Code
Ownership Schemes
Safer Rust Code
ORC Analyzer
Fully context-sensitive, flow-sensitive static analyzer that infers all possible ownership transfer relations between program variables.
A dedicated solver that transforms those relations into 0-1 integer linear constraints and finds solutions

```rust
pub struct node {
    pub key: i32,
    pub left: Option<Box<node>>,
    pub right: Option<Box<node>>,
}

impl Default for node {
    fn default() -> Self {
        Self {
            key: Default::default(),
            left: None,
            right: None,
        }
    }
}

// A utility function to create a new BST node
pub fn newNode(mut item: i32) -> Option<Box<node>> {
    let mut temp =
        Some(Box::new(<node as Default>::default()));
    (*temp.as_deref_mut().unwrap()).key = item;
    (*temp.as_deref_mut().unwrap()).right = None;
    (*temp.as_deref_mut().unwrap()).left = None;//(*temp).right;
    return temp;
}
// A utility function to do inorder traversal of BST
pub fn inorder(mut root: Option<&node>) {
    if !root.clone().is_none() {
        inorder((*root.clone().unwrap()).left.as_deref());
        println!("{:?}", (*root.as_deref().unwrap()).key);
        inorder((*root.clone().unwrap()).right.as_deref());
    } else {
        root = None;
    }
}

pub fn insert(mut node: Option<Box<node>>, mut key: i32)
 -> Option<Box<node>> {
    /* If the tree is empty, return a new node */
    if node.as_deref().is_none() {
        node = None;
    //     // free(node as *mut libc::c_void);
        return newNode(key)
    }
    /* Otherwise, recur down the tree */
    if key < (*node.as_deref().unwrap()).key {
        (*node.as_deref_mut().unwrap()).left = insert((*node.as_deref_mut().unwrap()).left.take(), key)
    } else { (*node.as_deref_mut().unwrap()).right = insert((*node.as_deref_mut().unwrap()).right.take(), key) }
    /* return the (unchanged) node pointer */
    return node;
}
 }
```

100% safety

SUMMARY

- Infer proper ownership schemes, thereby infer correct smart pointer types to help rewrite C programs
- Properly handle the ownership of function parameters and local variables
- Able to rewrite some fundamental data structures and their usages (typically singly linked lists, which prevail in C projects)
- In Progress: handle lifetime mechanism
  - Further enhance safety ratio
- In Progress: Evaluation with Huawei
  - Rosetta Stone, Fundamental Data Structure
  - Larger Projects (PtrDist, Busybox)
  - Prototype tool by the end of 2022!

THANK YOU