RefinedRust

A Type System for High-Assurance Verification of Rust Programs

PLDI 2024, Copenhagen

Lennard Gäher, Michael Sammler, Ralf Jung,
Robbert Krebbers, Derek Dreyer

MAX PLANCK INSTITUTE
FOR SOFTWARE SYSTEMS

ETH zürich

Radboud University

SIC Saarland Informatics Campus

RUST IS AN EXCITING LANGUAGE

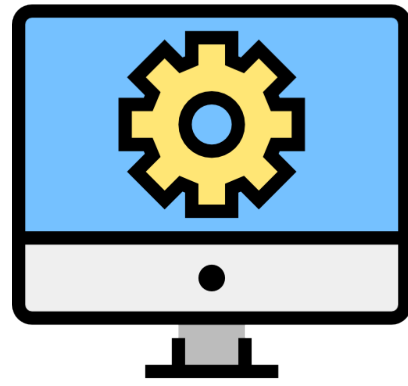
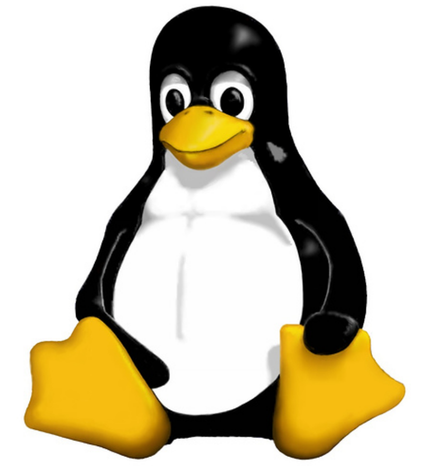
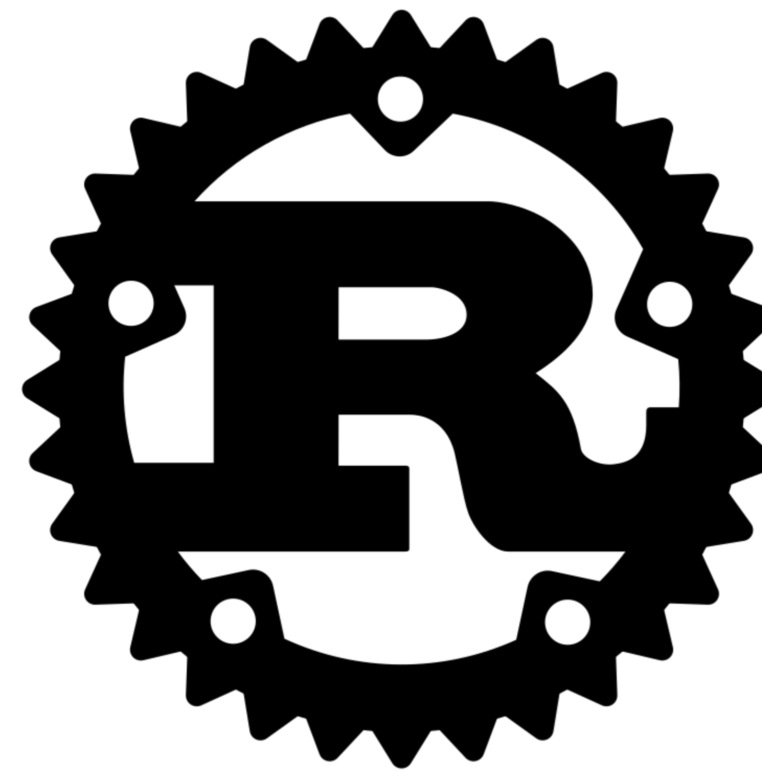


image: Flaticon.com

Systems programming
with zero-cost abstractions
for memory management

Growing ecosystem and
increasing popularity





Brings modern programming paradigms
to systems programming

```
x.iter().map(|x| x + 2).collect()
```

Aims to provide **memory safety for free**(*):

- no null-pointer accesses
- no use-after-free
- no data races
- …

(*) not for unsafe code

EXISTING RUST VERIFICATION TOOLS

We want to verify Rust programs!

Creusot [Denis et al. 2022]

Flux [Lehmann et al. 2023]

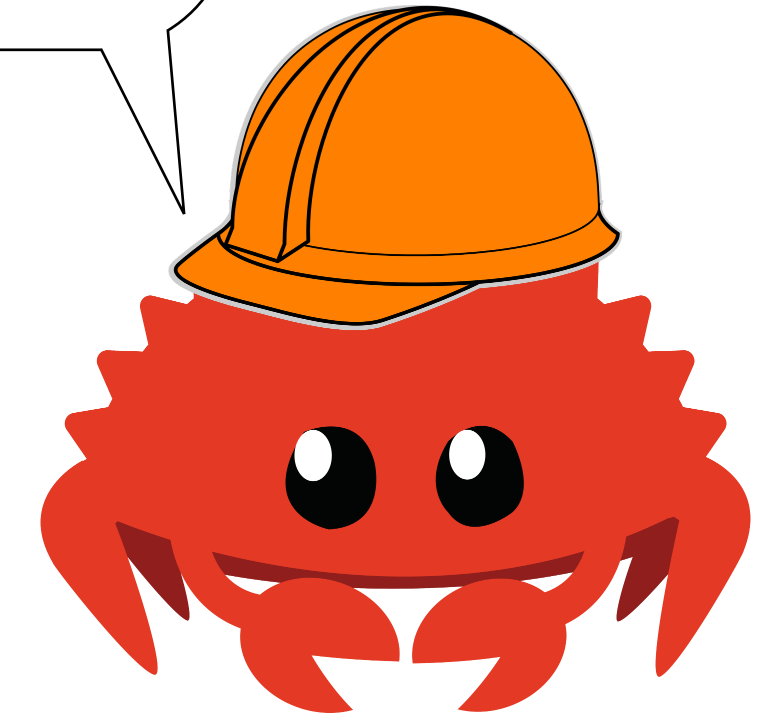
Aeneas [Ho et al. 2022]

RustHorn [Matsushita et al. 2020]

Prusti [Astrauskas et al. 2019]

Verus [Lattuada et al. 2023]

But what about unsafe Rust?



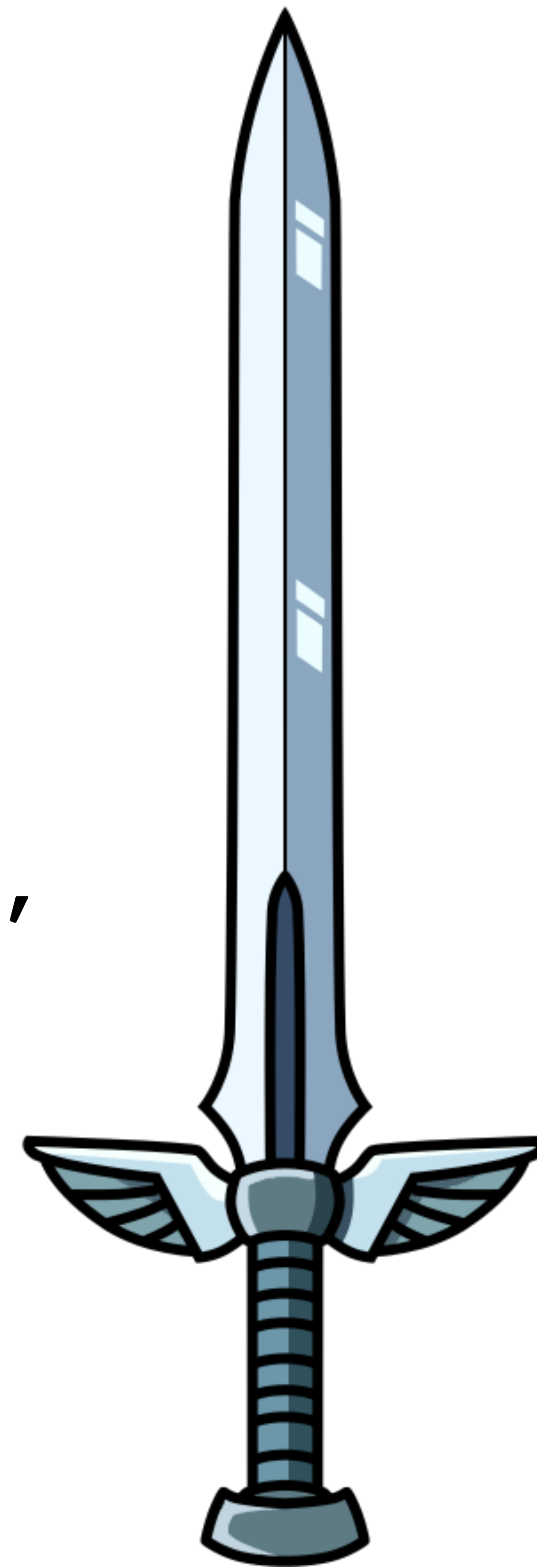
UNSAFE RUST IS A DOUBLE-EDGED SWORD

Unsafe Rust enables C-style raw **pointer manipulation**, but is **not guaranteed to be memory-safe**

Unsafe Rust **is crucial to the success of Rust** as it underlies some of the most **widely-used** Rust code, e.g. in `Vec`

```
unsafe {
    ptr::write(self.ptr().add(self.len),
        elem);
}
```

unchecked pointer arithmetic



Unsafe Rust is inherently **less well-behaved,** making **verification more complicated:**

- custom **ownership reasoning**
- an **accurate memory model**

INTRODUCING REFINEDRUST

RefinedRust is the first system that:

- verifies functional correctness of **both safe and unsafe** Rust code
- and generates **foundational proofs** in the Coq proof assistant

OUR APPROACH: TURNING RUSTBELT INTO A VERIFICATION TOOL

RustBelt [Jung et al., POPL 2018]

a semantic model for Rust

➔ handles Rust's reference types
➔ (manually) reason about safe encapsulation of unsafe code

+

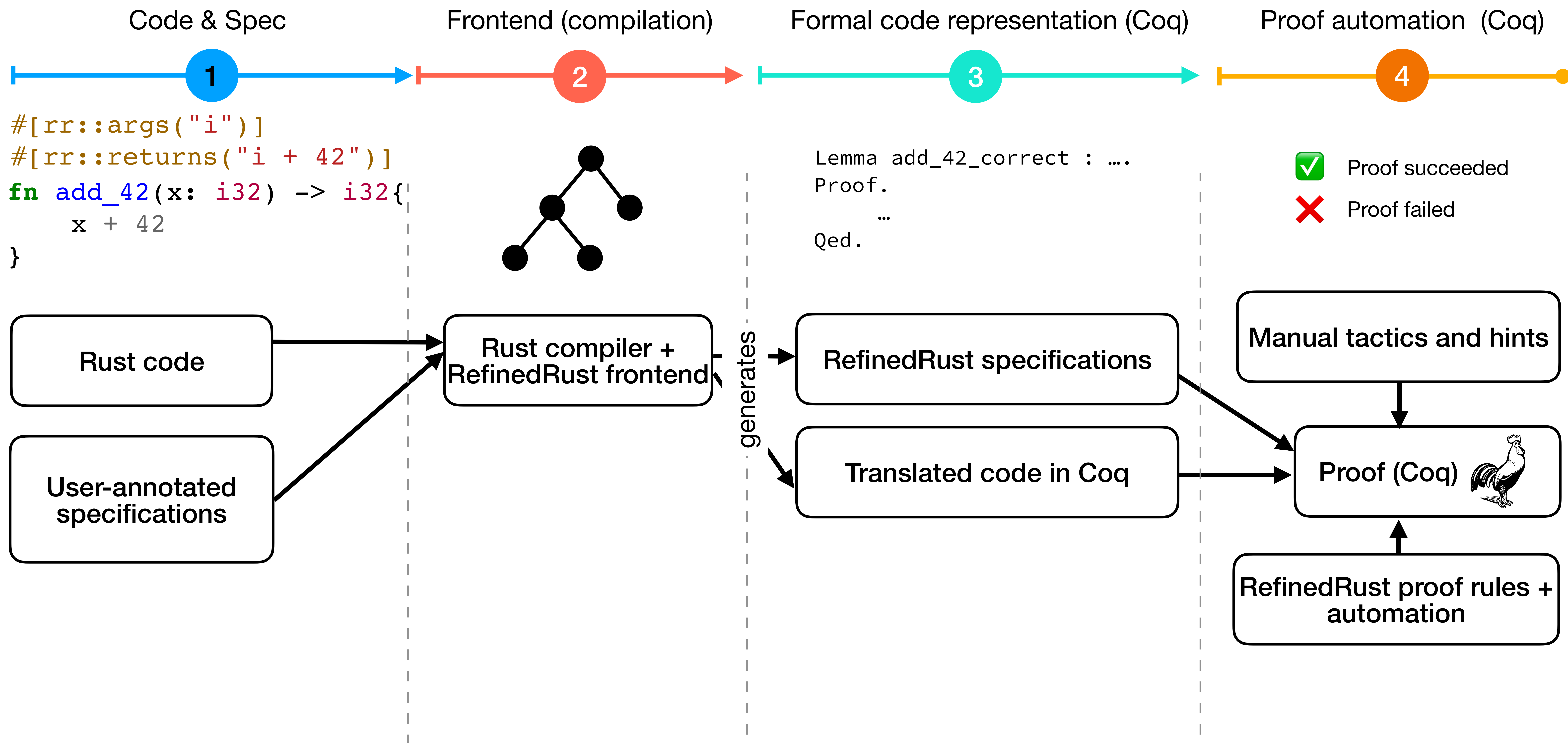
RefinedC [Sammler et al., PLDI 2021]

an ownership-based refinement type system for C

➔ refinement types for functional correctness reasoning
➔ ownership types to handle C code
➔ Lithium: separation logic automation

incompatible ownership systems, RefinedC-style refinements don't work for Rust

THE REFINEDRUST ARCHITECTURE



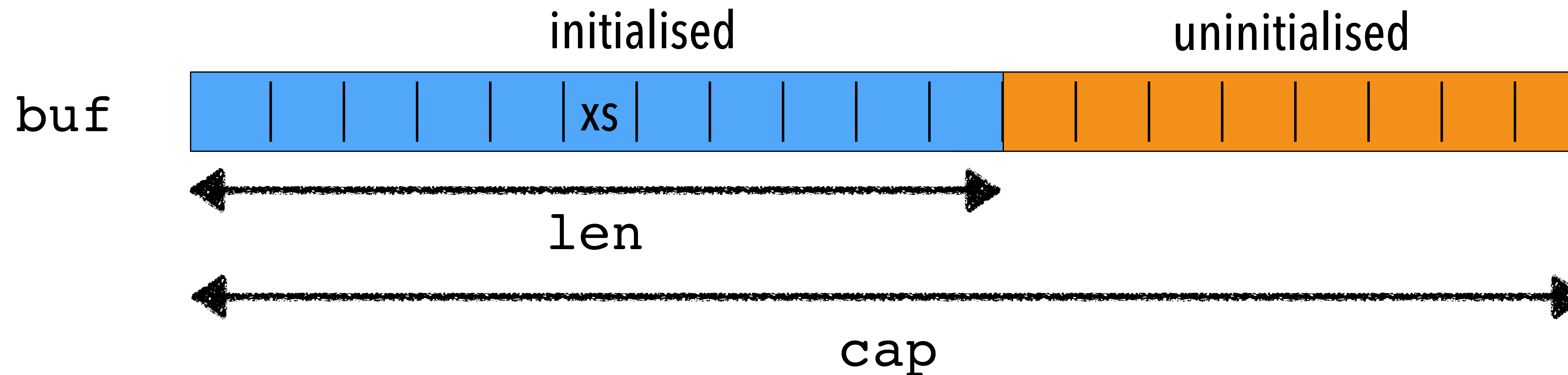
VERIFYING SAFE RUST CODE

```
/// Add 42 to the argument x and return the result.
#[rr::params("i" : "Z")]
#[rr::args("i")]
#[rr::requires("(i + 42) ∈ i32")]
#[rr::returns("i + 42")]
fn add_42(x: i32) -> i32 {
    x + 42
}
```

VERIFYING UNSAFE CODE WITH REFINEDRUST

VERIFYING VEC: MEMORY REPRESENTATION

```rust
pub struct Vec<T> {
    buf: *const T,
    len: usize,
    cap: usize,
}
```



First task: define **representation invariant** of `Vec`

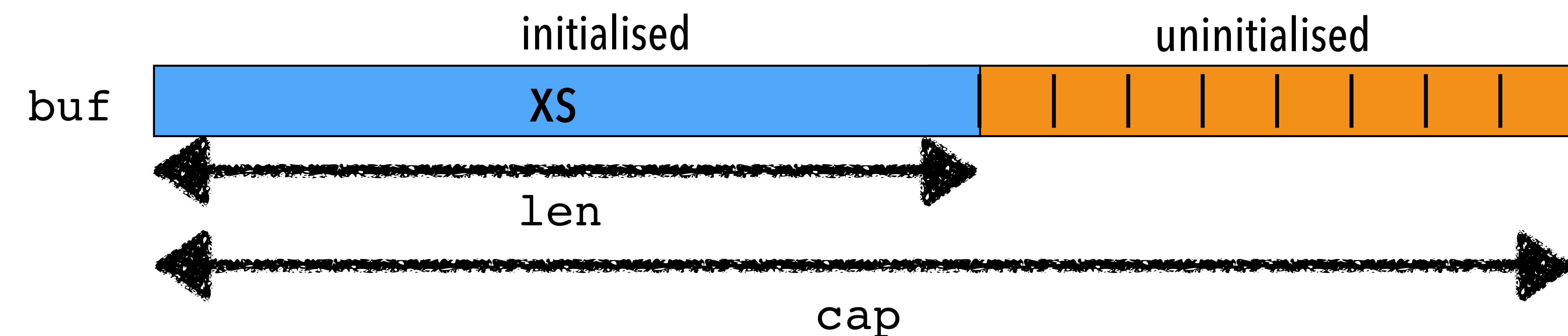
`Vec` just exposes the list `xs` of its initialised elements

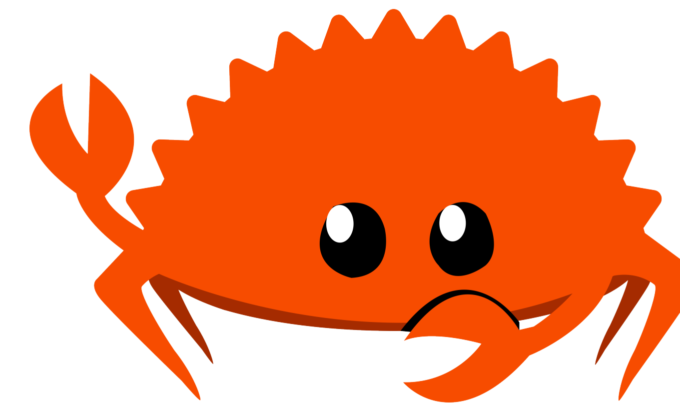
CUSTOM REPRESENTATION INVARIANTS

`Vec` just exposes the list `xs` of its initialised elements

```
#[rr::refined_by("xs" : "list (bor T)")]
#[rr::exists("cap", "len", "l", "els")]
#[rr::invariant(#type "l" : "els" @ "array_t (maybe_init T) cap")]
#[rr::invariant("∀ i, 0 ≤ i < len → els[i] = #(Some (xs[i]))")]
#[rr::invariant("∀ i, len ≤ i < cap → els[i] = #None")]
#[rr::invariant("len = length xs", "len ≤ cap")]
#[rr::invariant("size_of_array_in_bytes {st_of T} cap ≤ max_int isize_t")]
pub struct Vec<T> {
    #[rr::field("l")]
    buf: *const T,
    #[rr::field("len")]
    len: usize,
    #[rr::field("cap")]
    cap: usize,
}
```

Asserts **exclusive ownership**!





VERIFYING GET_MUT

```
#[rr::params("xs", "i"      )]
#[rr::args("(#(<#> xs)     )", "i")]
#[rr::requires("0 ≤ i < length xs")]
#[rr::exists("γi")]
#[rr::returns("(#(xs[i])     )")]
#[rr::resolve("γ": "(<#> xs)[i := *γi]")]
pub unsafe fn get_unchecked_mut(&mut self, index: usize) -> &mut T {
    unsafe {
        let p = self.ptr().add(index);
        let ret = &mut *p;
        ret
    }
}
```

"Return a mutable reference to the element of `self` at the given `index`"

Problem: how does the vector get changed if the i-th element is modified through the reference?

Borrow names **γ** communicate final values of references

How does this uphold the representation invariant of `Vec`? (ownership temporarily moved out)!

PRELIMINARY EVALUATION

We verified most of the Vec Rustonomicon implementation:

	Rust LOC	Translated LOC	Sideconds	Spec + manual proof	Verification time
Vec::new	6	8	23	1 + 0	18s
Vec::push	9	78	215	5 + 38	5min 12s
Vec::pop	8	56	121	4 + 17	2min 30s
Vec::get_unchecked_mut	7	32	69	6 + 11	2min
Vec::get_mut	8	49	108	7 + 29	1min 05s
Vec::get_unchecked	7	28	53	4 + 3	46s
Vec::get	8	49	93	3 + 0	48s
Total (RawVec + Vec)	**120 (14 functions)**	**950**	**>700**	**75 + 105**	**<6min (wall)**

(with ~80 lines of common manually proved Coq theory about lists)

Results:

- We can get a **good degree of automation in most cases**, but pure side conditions sometimes need help
- RefinedRust is **slower than tools for safe Rust code** (due to Coq, complex reasoning about unsafe)

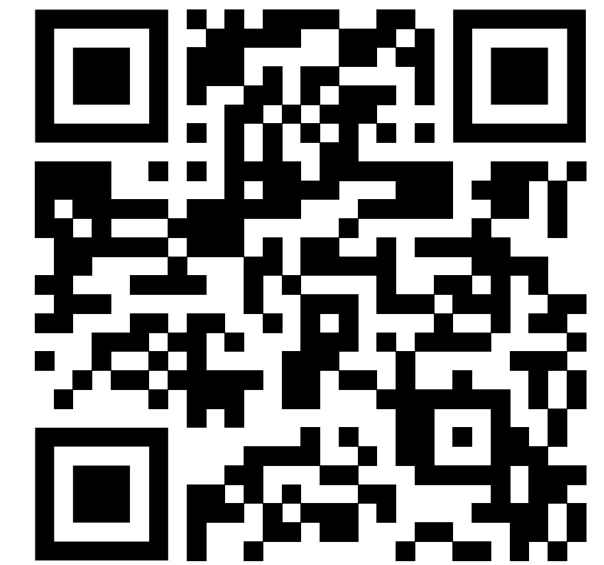
ONGOING WORK: SCALING UP REFINEDRUST

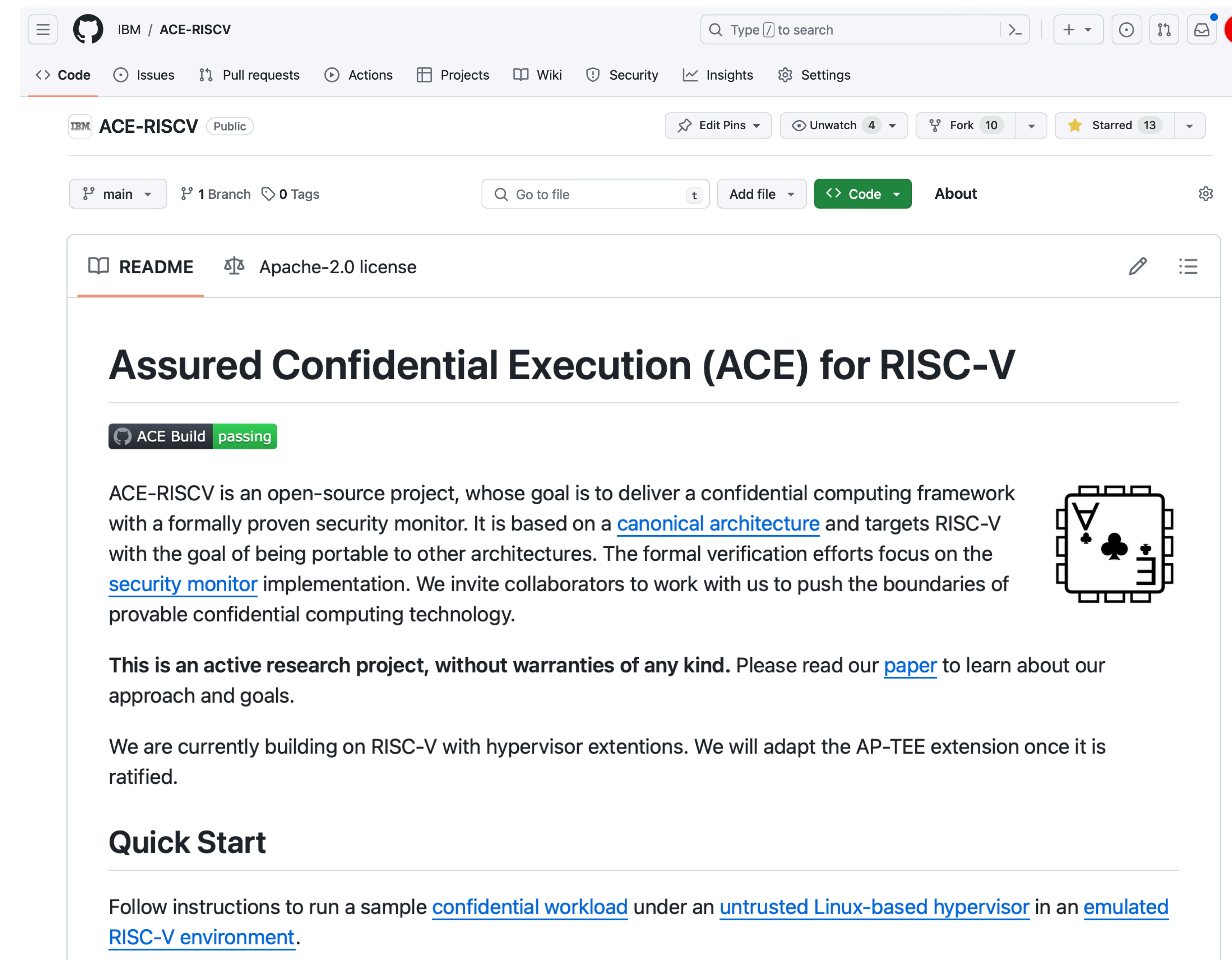
- Adding interior mutability (with Vincent Lafeychine)

- Making RefinedRust more practical & scalable
- Trying to verify memory safety & security of a practical security monitor implementation

(with IBM Research: Wojciech Ozga, Avi Shinnar, Elaine Palmer, Michael Le, Guerney Hunt, Silvio Dragone)

https://github.com/
IBM/ACE-RISCV





Confidential VM Extension (CoVE) for Confidential Computing

Version 0.6, 4/2024: This document is under development. Expect potential changes. Visit http://riscv.org/spec-state for further details.

REFINEDRUST: WHAT ELSE IS IN THE PAPER

Place types to capture intermediate states of the Rust type system
➔ enabled by a novel extension of RustBelt's lifetime logic with "pinned borrows"

Borrow variables to communicate value updates of Rust's mutable references
➔ inspired by RustHornBelt's prophecy variables [Matsushita et al., PLDI 2022]

Syntactic types to enable layout-generic proofs
➔ verification is parameterised by the Rust layout algorithm

plv.mpi-sws.org/refinedrust/

HOW TO CAPTURE INTERMEDIATE STATES OF THE TYPE SYSTEM?

```
let x: (i32, i32) = (1, 2);

let xr: &'a mut i32 = &mut x.0;

*xr = 42;

assert!(x.0 == 42);
```

What is the type assignment of x here?
We need to track that the first component is borrowed…

x.0 is inaccessible, as it is borrowed for lifetime 'a

Key idea: Use **place types** to describe intermediate states of the Rust type system
(e.g. induced by the borrow checker)

PLACE TYPES DESCRIBE INTERMEDIATE STATES OF THE TYPE SYSTEM

type assignment

```
let x: (i32, i32) = (1, 2);
  { x ◁ place (i32, i32) }
let xr: &mut i32 = &mut x.0;
 { xr ◁ place (&mut 'a i32) *
   x ◁ (blocked 'a i32, place i32) }
*xr = 42;
 { xr ◁ place (&mut 'a i32) *
   x ◁ (blocked 'a i32, place i32) }
assert!(x.0 == 42);
 { x ◁ place (i32, i32)}
```

x is a stack variable containing a tuple

The first component of x is inaccessible until the reference's lifetime ‘ a ends

‘ a implicitly ends

Once ‘ a ends, x is fully accessible again

(omitting refinements)

PLACE TYPES

Place types also help us to describe the **intermediate states in unsafe cases** the Rust type system does not support!

```
pub unsafe fn get_unchecked_mut(&mut self, index: usize) -> &mut T {
    unsafe {
        let p = self.ptr().mut_add(index);
        let ret = &mut *p;
        ret
    }
}
```

…and place types are also used to **describe unfolded invariants on datatypes** (see the paper for details)

REFINEDRUST: WHAT ELSE IS IN THE PAPER

Place types to capture intermediate states of the Rust type system
➔ enabled by a novel extension of RustBelt's lifetime logic with "pinned borrows"

Borrow variables to communicate value updates of Rust's mutable references
➔ inspired by RustHornBelt's prophecy variables [Matsushita et al., PLDI 2022]

Syntactic types to enable layout-generic proofs
➔ verification is parameterised by the Rust layout algorithm

plv.mpi-sws.org/refinedrust/