RefinedRust

Towards high-assurance verification of unsafe Rust programs

Rust Verification Workshop 2023, Paris
23.04.2023

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

MAX PLANCK INSTITUTE
FOR SOFTWARE SYSTEMS

ETH zürich

Radboud University

with generous funding from Amazon AWS

EXISTING RUST VERIFICATION TOOLS

We want to functionally verify Rust programs!

EXISTING RUST VERIFICATION TOOLS

We want to functionally verify Rust programs!

Prusti [Astrauskas et al. 2019]

EXISTING RUST VERIFICATION TOOLS

We want to functionally verify Rust programs!

RustHorn [Matsushita et al. 2020]

Prusti [Astrauskas et al. 2019]

EXISTING RUST VERIFICATION TOOLS

We want to functionally verify Rust programs!

Creusot [Denis et al. 2022]

RustHorn [Matsushita et al. 2020]

Prusti [Astrauskas et al. 2019]

EXISTING RUST VERIFICATION TOOLS

We want to functionally verify Rust programs!

Creusot [Denis et al. 2022]

Aeneas [Ho et al. 2022]

RustHorn [Matsushita et al. 2020]

Prusti [Astrauskas et al. 2019]

EXISTING RUST VERIFICATION TOOLS

We want to functionally 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]

EXISTING RUST VERIFICATION TOOLS

We want to functionally verify Rust programs!

Creusot [Denis et al. 2022]

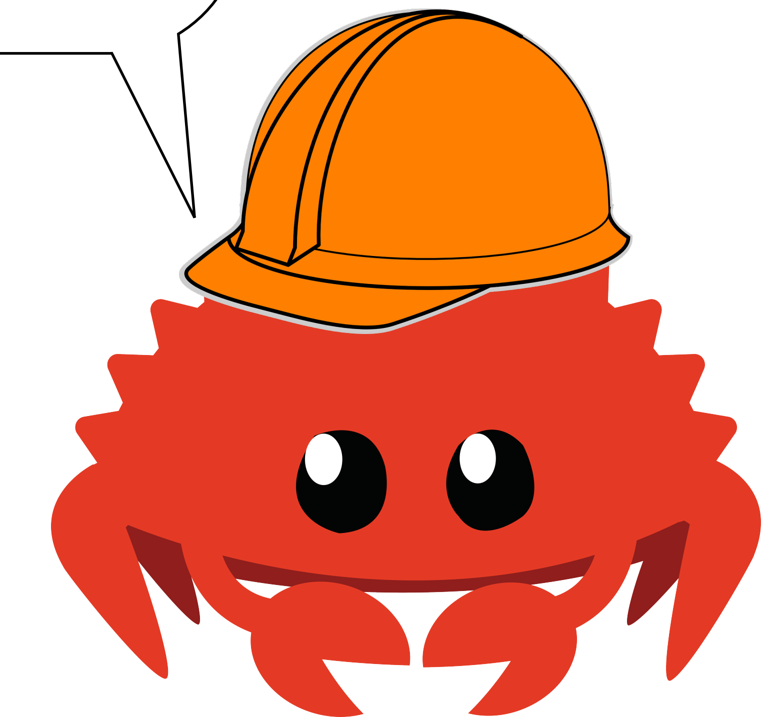
Flux [Lehmann et al. 2023]

Aeneas [Ho et al. 2022]

RustHorn [Matsushita et al. 2020]

Prusti [Astrauskas et al. 2019]

But what about unsafe Rust?

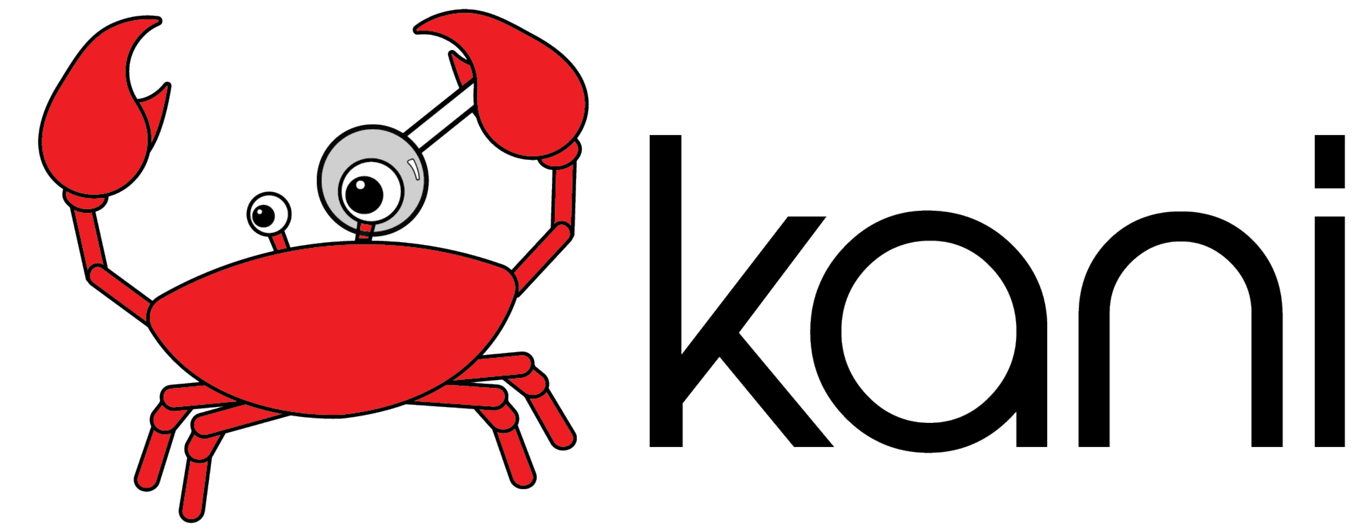


HOW DO WE VERIFY *UNSAFE* CODE?

RustBelt [Jung et al. 2019] /
RustHornBelt [Matsushita et al. 2022]

- ➔ a semantic model for Rust
- ➔ used to prove safety/functional correctness of several of Rust's core libraries

Kani [Kani Developers 2022]



- ➔ a model checker for Rust
- ➔ scalable automatic verification

MOST STANDARD LIBRARY CODE RELIES ON *UNSAFE*...

1.0.0 · source · [−]

```
pub struct Vec<T, A = Global>
where
    A: Allocator,
{ /* private fields */ }
```

[−] A contiguous growable array type, written as `Vec<T>`, short for 'vector'.

[−] `pub fn push(&mut self, value: T)` source

Appends an element to the back of a collection.

Panics

Panics if the new capacity exceeds `isize::MAX` bytes.

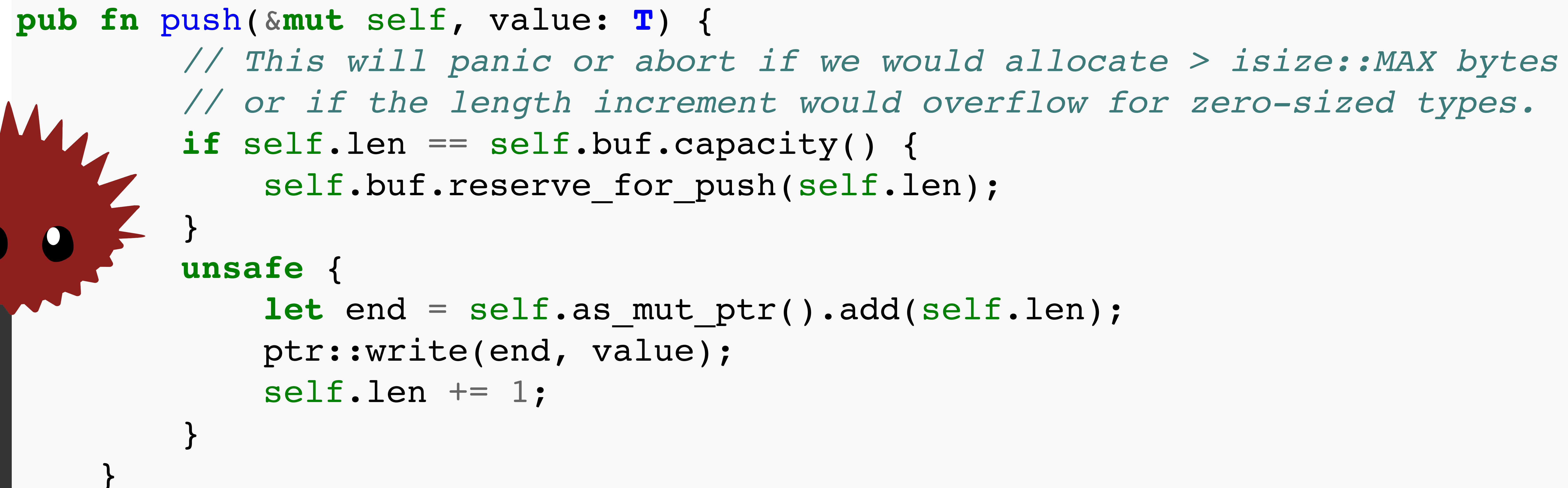
Examples

```
let mut vec = vec![1, 2];
vec.push(3);
assert_eq!(vec, [1, 2, 3]);
```

MOST STANDARD LIBRARY CODE RELIES ON *UNSAFE*...

1.0.0 · source · [-]

```
pub struct Vec<T, A = Global>
where
```

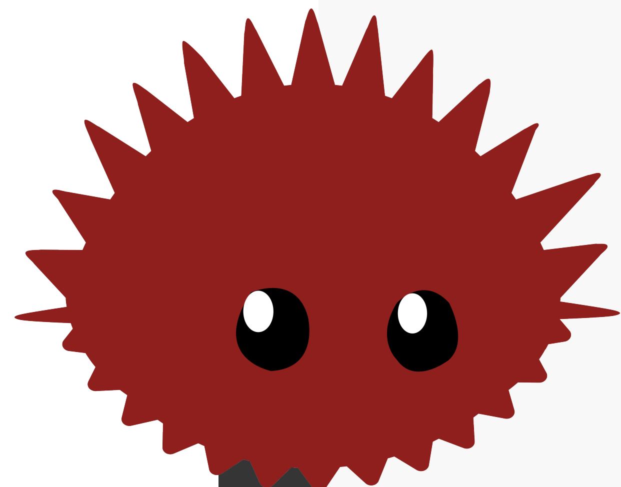



```
let mut vec = vec![1, 2];
vec.push(3);
assert_eq!(vec, [1, 2, 3]);
```

MOST STANDARD LIBRARY CODE RELIES ON *UNSAFE*...

1.0.0 · source · [-]

```
pub struct Vec<T, A = Global>
where
```



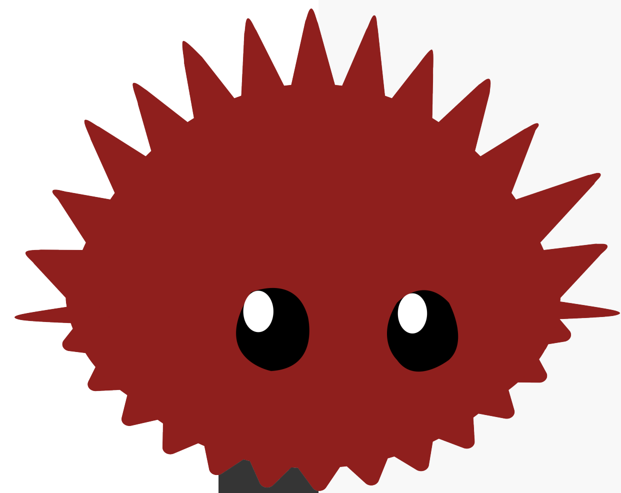
```
pub fn push(&mut self, value: T) {
        // This will panic or abort if we would allocate > isize::MAX bytes
        // or if the length increment would overflow for zero-sized types.
        if self.len == self.buf.capacity() {
            self.buf.reserve_for_push(self.len);
        }
        unsafe {
            let end = self.as_mut_ptr().add(self.len);
            ptr::write(end, value);
            self.len += 1;
        }
    }
```

```
let mut vec = vec![1, 2];
vec.push(3);
assert_eq!(vec, [1, 2, 3]);
```

MOST STANDARD LIBRARY CODE RELIES ON *UNSAFE*...

1.0.0 · source · [−]

```
pub struct Vec<T, A = Global>
where
```



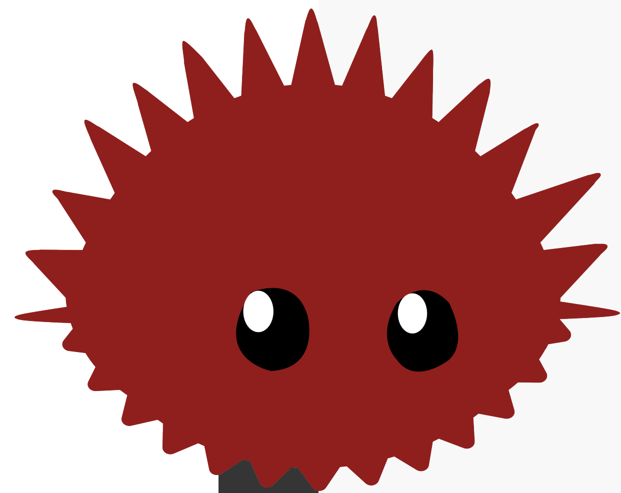
```
pub fn push(&mut self, value: T) {
        // This will panic or abort if we would allocate > isize::MAX bytes
        // or if the length increment would overflow for zero-sized types.
        if self.len == self.buf.capacity() {
            self.buf.reserve_for_push(self.len);
        }
        unsafe {
            let end = self.as_mut_ptr().add(self.len);
            ptr::write(end, value);
            self.len += 1;
        }
    }
```

```
let mut vec = vec![1, 2];
vec.push(3);
assert_eq!(vec, [1, 2, 3]);
```

MOST STANDARD LIBRARY CODE RELIES ON *UNSAFE*...

1.0.0 · source · [-]

```
pub struct Vec<T, A = Global>
where
```



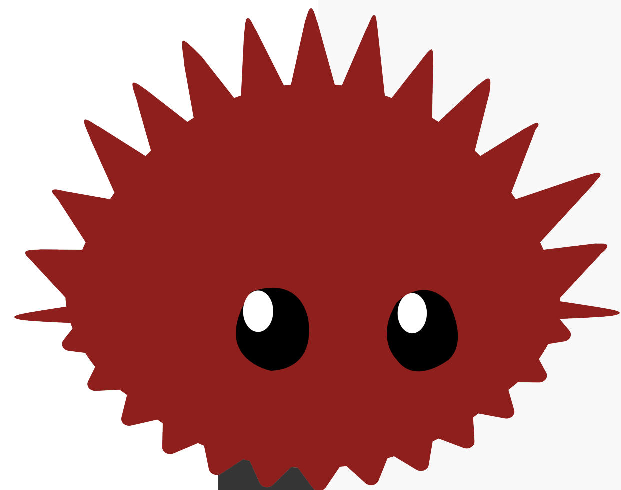
```
pub fn push(&mut self, value: T) {
        // This will panic or abort if we would allocate > isize::MAX bytes
        // or if the length increment would overflow for zero-sized types.
        if self.len == self.buf.capacity() {
            self.buf.reserve_for_push(self.len);
        }
        unsafe {
            let end = self.as_mut_ptr().add(self.len);
            ptr::write(end, value);
            self.len += 1;
        }
    }
```

```
let mut vec = vec![1, 2];
vec.push(3);
assert_eq!(vec, [1, 2, 3]);
```

MOST STANDARD LIBRARY CODE RELIES ON *UNSAFE*...

1.0.0 · source · [-]

```
pub struct Vec<T, A = Global>
where
```



```
pub fn push(&mut self, value: T) {
        // This will panic or abort if we would allocate > isize::MAX bytes
        // or if the length increment would overflow for zero-sized types.
        if self.len == self.buf.capacity() {
            self.buf.reserve_for_push(self.len);
        }
        unsafe {
            let end = self.as_mut_ptr().add(self.len);
            ptr::write(end, value);
            self.len += 1;
        }
    }
```

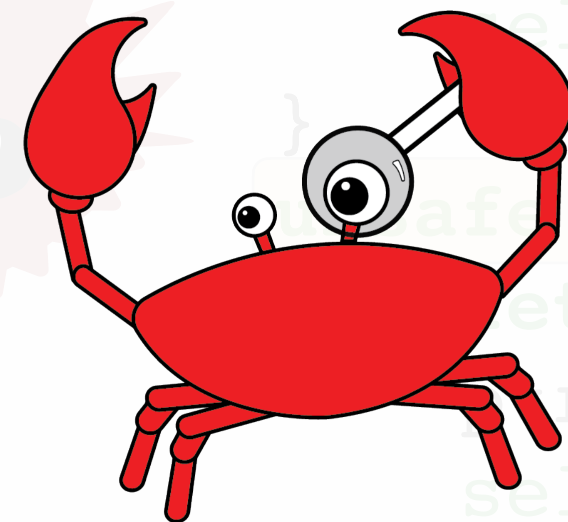
```
let mut vec = vec![1, 2];
vec.push(3);
assert_eq!(vec, [1, 2, 3]);
```

MOST STANDARD LIBRARY CODE RELIES ON *UNSAFE*...

fully manual verification ☹️

MOST STANDARD LIBRARY CODE RELIES ON *UNSAFE*...

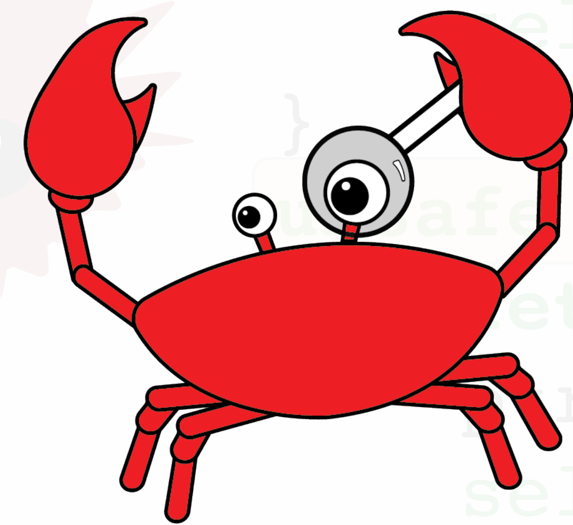
fully manual verification ☹️



no compositional specifications ☹️

MOST STANDARD LIBRARY CODE RELIES ON *UNSAFE*…

fully manual verification ☹️



no compositional specifications ☹️

Can we get a compositional and automated verification tool for unsafe Rust?

A PRACTICAL VERIFICATION TOOL FOR UNSAFE RUST?

We want a deductive verification tool with:

✓ support for common **unsafe** operations

✓ a reasonable degree of **automation**

✓ high-assurance **foundational** verification (e.g. in Coq)

TURNING RUSTBELT INTO A VERIFICATION TOOL

RustBelt [Jung et al. 2019]

a semantic model for Rust

➔ shows how to handle Rust's reference types

➔ allows to verify safe encapsulation of unsafe code

TURNING RUSTBELT INTO A VERIFICATION TOOL

RustBelt [Jung et al. 2019]

+

RefinedC [Sammler et al. 2021]

a semantic model for Rust

- ➔ shows how to handle Rust's reference types
- ➔ allows to verify safe encapsulation of unsafe code

an ownership-based refinement type system for C

- ➔ refinement types for functional correctness reasoning
- ➔ naturally handles "unsafe code" with ownership types
- ➔ Lithium: efficiently automatable separation logic fragment

INTRODUCING REFINEDRUST

Goal: verify functional correctness & UB-freedom & panic-freedom

INTRODUCING REFINEDRUST

Goal: verify functional correctness & UB-freedom & panic-freedom

Radium **operational semantics** for Rust based on RefinedC

INTRODUCING REFINEDRUST

Goal: verify functional correctness & UB-freedom & panic-freedom

An **automatic translation** scheme from Rust into Radium

Radium **operational semantics** for Rust based on RefinedC

INTRODUCING REFINEDRUST

Goal: verify functional correctness & UB-freedom & panic-freedom

An **automatic translation** scheme from Rust into Radium

Radium **operational semantics** for Rust based on RefinedC

Refinement type system with semantic model inspired by RustBelt

INTRODUCING REFINEDRUST

Goal: verify functional correctness & UB-freedom & panic-freedom

An **automatic translation** scheme from Rust into Radium

Proof automation based on RefinedC's Lithium engine

Radium **operational semantics** for Rust based on RefinedC

Refinement type system with semantic model inspired by RustBelt

CHALLENGES IN REFINEDRUST

Lift RustBelt's limitations

✓ allow automatic translation from Rust

✓ handle more borrow patterns

- develop new place-based type system
- extend RustBelt's lifetime logic

CHALLENGES IN REFINEDRUST

Lift RustBelt's limitations

- ✓ allow automatic translation from Rust
- ✓ handle more borrow patterns
- develop new place-based type system
- extend RustBelt's lifetime logic

Equip RefinedC with references

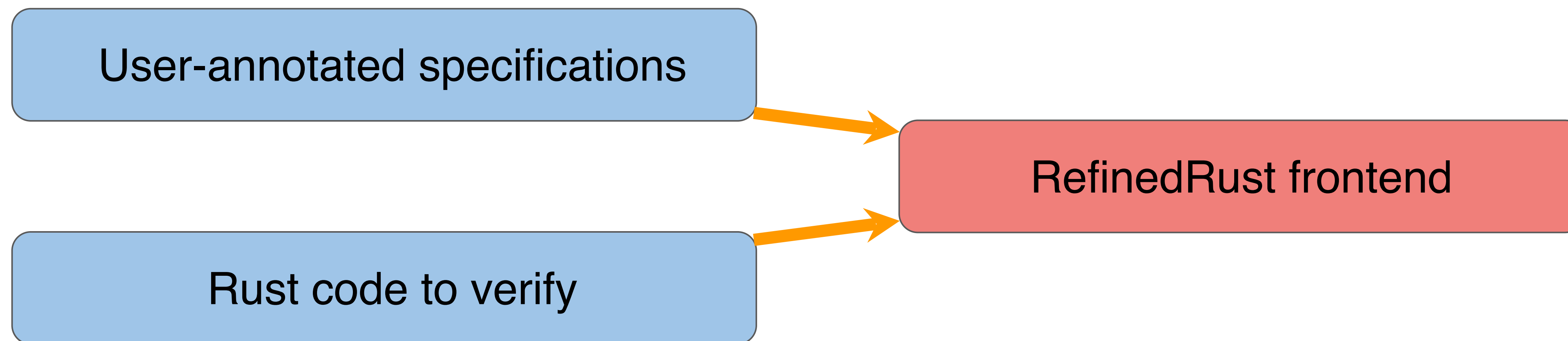
- ✓ need to rethink RefinedC's refinement model
- ✓ develop automation for Rust specifics

THE REFINEDRUST ARCHITECTURE

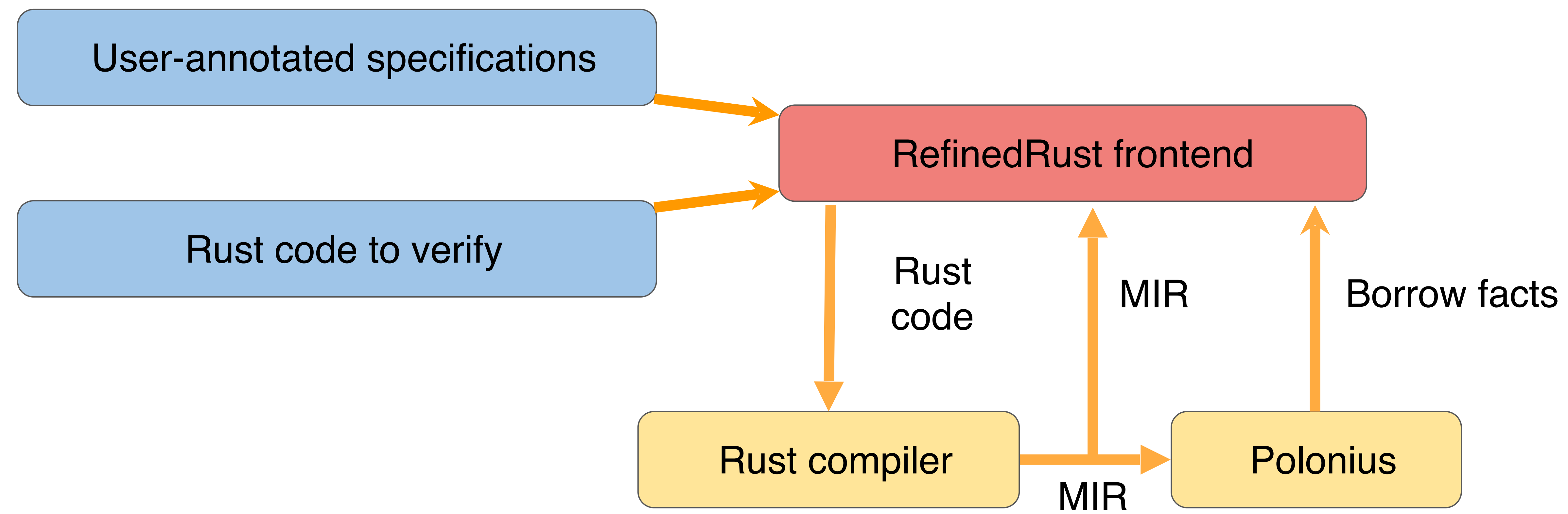
User-annotated specifications

Rust code to verify

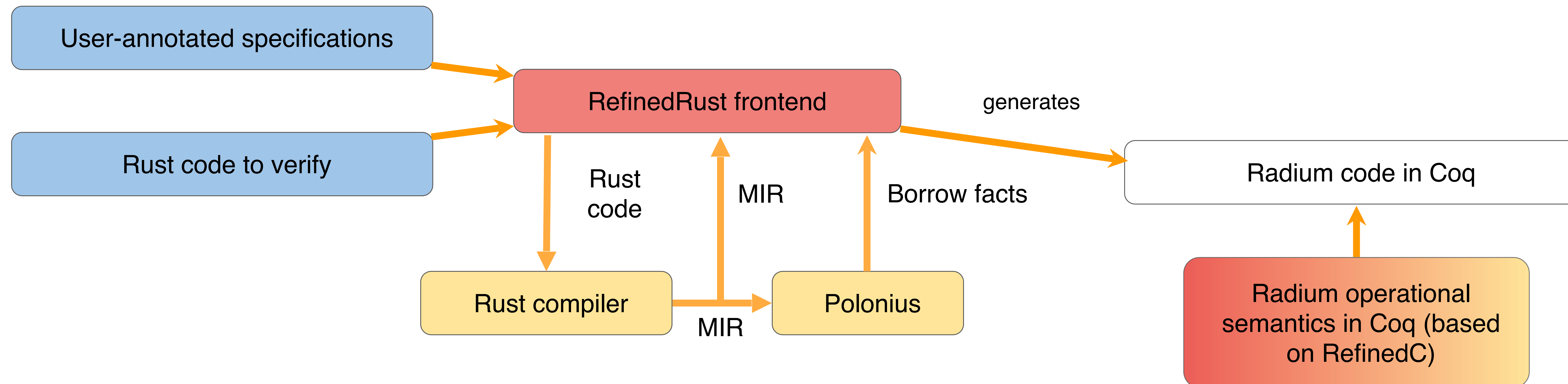
THE REFINEDRUST ARCHITECTURE



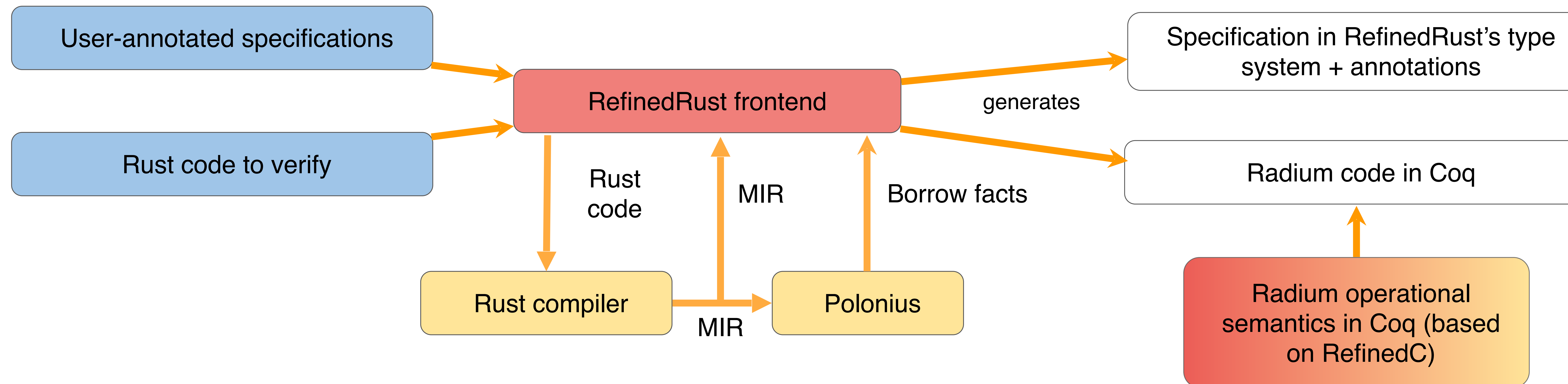
THE REFINEDRUST ARCHITECTURE



THE REFINEDRUST ARCHITECTURE

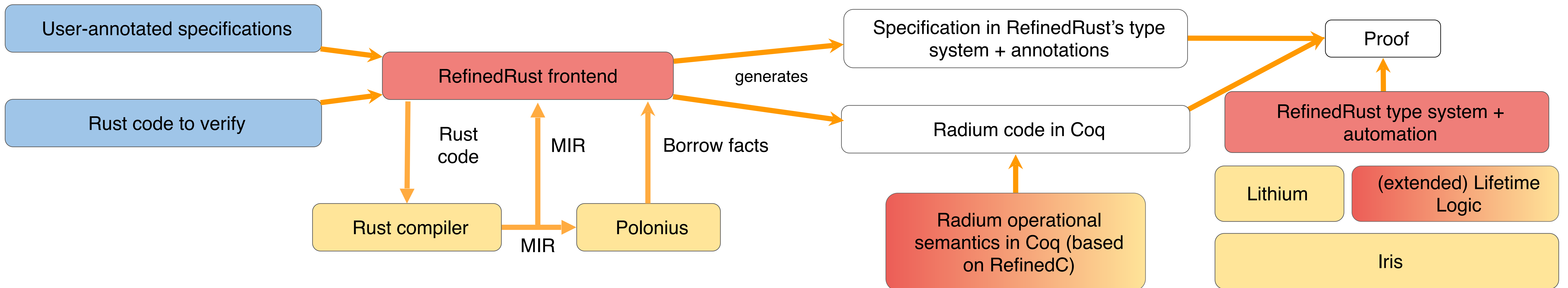


THE REFINEDRUST ARCHITECTURE

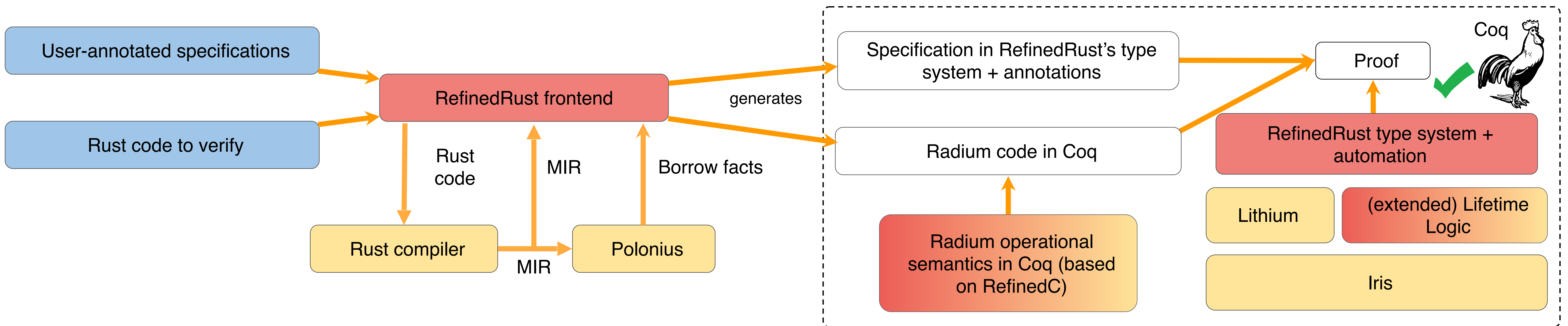


Radium operational semantics in Coq (based on RefinedC)

THE REFINEDRUST ARCHITECTURE



THE REFINEDRUST ARCHITECTURE



USING REFINEDRUST FOR VERIFYING SAFE RUST

MUTABLE REFERENCES IN REFINEDRUST

```
fn mut_ref_add_42(x : &mut i32) {
    *x += 42;
}
```

MUTABLE REFERENCES IN REFINEDRUST

```
#[rr::params("x"      )]

fn mut_ref_add_42(x : &mut i32) {
    *x += 42;
}
```

MUTABLE REFERENCES IN REFINEDRUST

```
#[rr::params("x"        )]
#[rr::args("             " @ "&mut {'a} (int i32)")]

fn mut_ref_add_42(x : &mut i32) {
    *x += 42;
}
```


L. Gäher, M. Sammler, R. Jung, R. Krebbers, D. Dreyer

MUTABLE REFERENCES IN REFINEDRUST

refined by *(current value, borrow variable)*

```
#[rr::params("x", "γ")]
#[rr::args("(#x, γ)" @ "&mut {'a} (int i32)")]

fn mut_ref_add_42(x : &mut i32) {
    *x += 42;
}
```

MUTABLE REFERENCES IN REFINEDRUST

Borrow variables communicate final values

(inspired by prophecy variables [Matsushita et al. 2020])

obtain observation on final value of γ

```
#[rr::params("x", "γ")]
#[rr::args("(#x, γ)" @ "&mut {'a} (int i32)")]
#[rr::ensures("Obs γ (x + 42)")]
fn mut_ref_add_42(x : &mut i32) {
    *x += 42;
}
```

MUTABLE REFERENCES IN REFINEDRUST

Borrow variables communicate final values

(inspired by prophecy variables [Matsushita et al. 2020])

```
#[rr::params("x", "γ")]
#[rr::args("(#x, γ)" @ "&mut {'a} (int i32)")]
#[rr::ensures("Obs γ (x + 42)")]
fn mut_ref_add_42(x : &mut i32) {
    *x += 42;
}
```

MUTABLE REFERENCES IN REFINEDRUST

Borrow variables communicate final values

(inspired by prophecy variables [Matsushita et al. 2020])

```
#[rr::params("x", "γ")]
#[rr::args("(#x, γ)" @ "&mut {'a} (int i32)")]
#[rr::ensures("Obs γ (x + 42)")]
fn mut_ref_add_42(x : &mut i32) {
    *x += 42;
}
```

Will this specification work?

MUTABLE REFERENCES IN REFINEDRUST

```
#[rr::params("x", "γ")]
#[rr::args("(#x, γ)" @ "&mut {'a} (int i32)")]
#[rr::requires("⌜x + 42 ∈ i32⌝")]
#[rr::ensures("Obs γ (x + 42)")]
fn mut_ref_add_42(x : &mut i32) {
    *x += 42;
}
```

MUTABLE REFERENCES IN REFINEDRUST

```
#[rr::params("x", "γ")]
#[rr::args("(#x, γ)" @ "&mut {'a} (int i32)")]
#[rr::requires("⌜x + 42 ∈ i32⌝")]
#[rr::ensures("Obs γ (x + 42)")]
fn mut_ref_add_42(x : &mut i32) {
    *x += 42;
}
```

MUTABLE REFERENCES IN REFINEDRUST

Types can be inferred from Rust types! (for safe Rust)

```
#[rr::params("x", "γ")]
#[rr::args("(#x, γ)")]
#[rr::requires("⌜x + 42 ∈ i32⌝")]
#[rr::ensures("Obs γ (x + 42)")]
fn mut_ref_add_42(x : &mut i32) {
    *x += 42;
}
```

A CLIENT FOR `MUT_REF_ADD_42`

```
#[rr::returns("()")]
fn mut_ref_add_client() {
    let mut z = 1;
    let zr = &mut z;
    mut_ref_add_42(zr);
    assert!(z == 43);
}
```


L. Gäher, M. Sammler, R. Jung, R. Krebbers, D. Dreyer

A CLIENT FOR MUT_REF_ADD_42

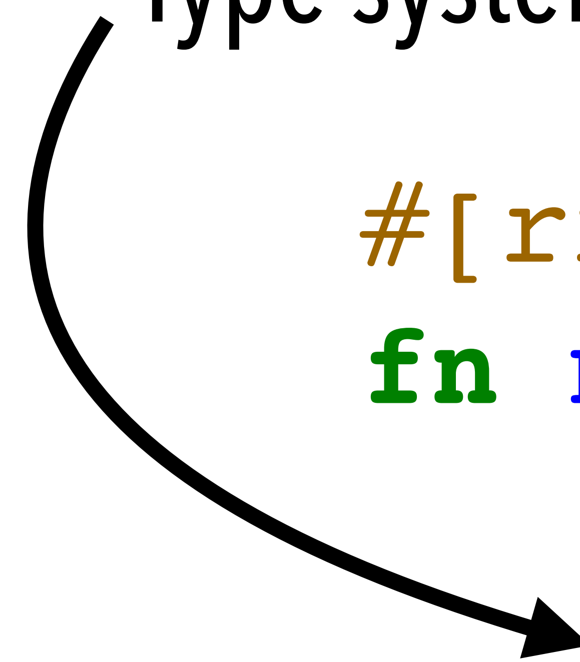
```
#[rr::returns("()")]
fn mut_ref_add_client() {
    let mut z = 1;
    let zr = &mut z;
    mut_ref_add_42(zr);
    assert!(z == 43);
}
```

A CLIENT FOR MUT_REF_ADD_42

Type system needs to track that z is borrowed

```
#[rr::returns("()")]
fn mut_ref_add_client() {
    let mut z = 1;
    let zr = &mut z;
    mut_ref_add_42(zr);
    assert!(z == 43);
}
```

A CLIENT FOR `MUT_REF_ADD_42`



Type system needs to track that `z` is borrowed

```
#[rr::returns("()")]
fn mut_ref_add_client() {
    let mut z = 1;
    let zr = &mut z;
    mut_ref_add_42(zr);
    assert!(z == 43);
}
```

A CLIENT FOR `MUT_REF_ADD_42`

Type system needs to track that `z` is borrowed

```
#[rr::returns("()")]
fn mut_ref_add_client() {
    let mut z = 1;
    let zr = &mut z;
    mut_ref_add_42(zr);
    assert!(z == 43);
}
```

Type system needs recombine observation `Obs`

VERIFYING UNSAFE CODE WITH REFINEDRUST

VERIFYING VEC: MEMORY REPRESENTATION

```
pub struct Vec<T> {
    buf: RawVec<T>,
    len: usize,
}
```

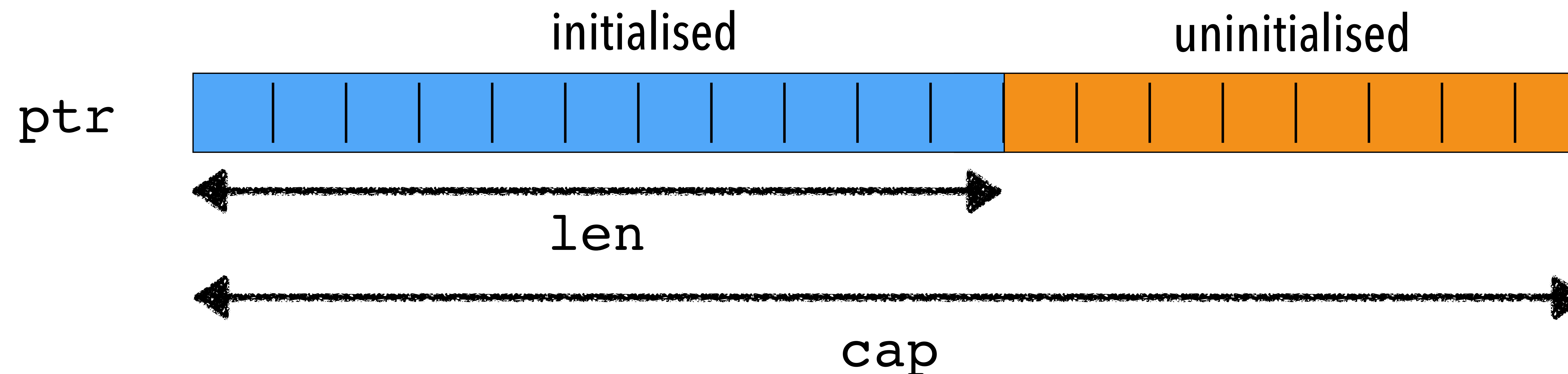
```
pub struct RawVec<T> {
    ptr: *const T,
    cap: usize,
    _marker: PhantomData<T>,
}
```

(we consider a variant of the Rustonomicon Vec implementation)

VERIFYING VEC: MEMORY REPRESENTATION

```
pub struct Vec<T> {
    buf: RawVec<T>,
    len: usize,
}
```

```
pub struct RawVec<T> {
    ptr: *const T,
    cap: usize,
    _marker: PhantomData<T>,
}
```

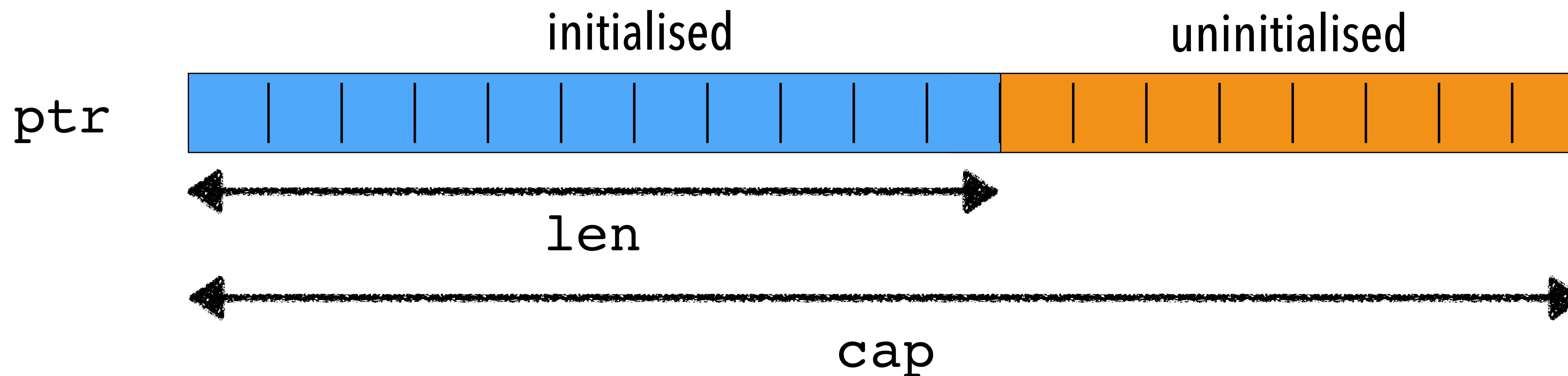


(we consider a variant of the Rustonomicon Vec implementation)

VERIFYING VEC: MEMORY REPRESENTATION

```
pub struct Vec<T> {
    buf: RawVec<T>,
    len: usize,
}
```

```
pub struct RawVec<T> {
    ptr: *const T,
    cap: usize,
    _marker: PhantomData<T>,
}
```



First task: define logical representation of `Vec`

(we consider a variant of the Rustonomicon Vec implementation)

CUSTOM REPRESENTATION INVARIANTS

`RawVec` just exposes the location and capacity

```
#[rr::refined_by("(l, cap)" : "(loc * nat)")]
#[rr::invariant(#own "freeable l (size_of_array_in_bytes {st_of T} cap)")]
pub struct RawVec<T> {
    #[rr::field("l")]
    ptr: *const T,
    #[rr::field("cap")]
    cap: usize,
    #[rr::field("tt")]
    _marker: PhantomData<T>,
}
```

(simplified invariant, does not handle the case that `T` is a ZST)

CUSTOM REPRESENTATION INVARIANTS

`RawVec` just exposes the location and capacity

```
#[rr::refined_by("(l, cap)" : "(loc * nat)")]
#[rr::invariant(#own "freeable l (size_of_array_in_bytes {st_of T} cap)")]
pub struct RawVec<T> {
    #[rr::field("l")]
    ptr: *const T,
    #[rr::field("cap")]
    cap: usize,
    #[rr::field("tt")]
    _marker: PhantomData<T>,
}
```

(simplified invariant, does not handle the case that `T` is a ZST)

CUSTOM REPRESENTATION INVARIANTS

`RawVec` just exposes the location and capacity

```
#[rr::refined_by("(l, cap)" : "(loc * nat)")]
#[rr::invariant(#own "freeable l (size_of_array_in_bytes {st_of T} cap)")]
pub struct RawVec<T> {
    #[rr::field("l")]
    ptr: *const T,
    #[rr::field("cap")]
    cap: usize,
    #[rr::field("tt")]
    _marker: PhantomData<T>,
}
```

(simplified invariant, does not handle the case that `T` is a ZST)

CUSTOM REPRESENTATION INVARIANTS

`RawVec` just exposes the location and capacity

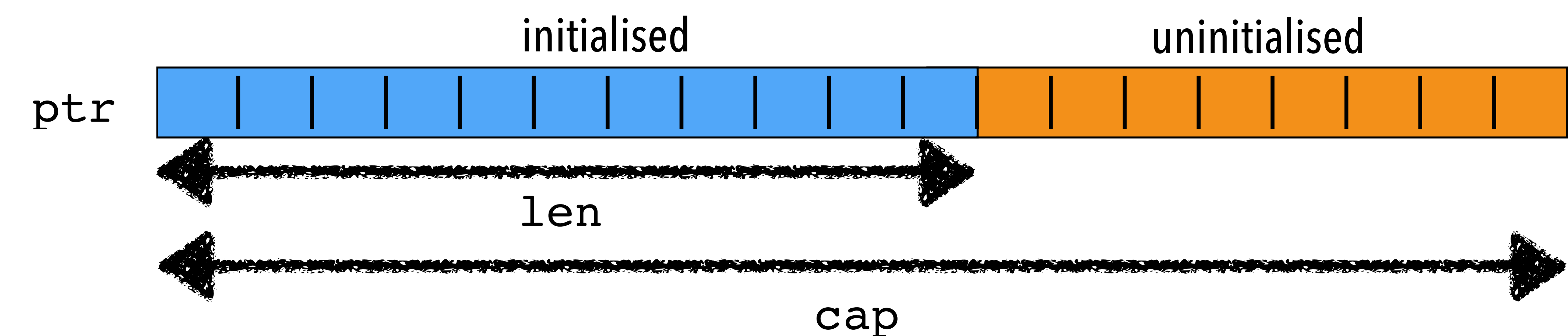
```
#[rr::refined_by("(l, cap)" : "(loc * nat)")]
#[rr::invariant(#own "freeable l (size_of_array_in_bytes {st_of T} cap)")]
pub struct RawVec<T> {
    #[rr::field("l")]
    ptr: *const T,
    #[rr::field("cap")]
    cap: usize,
    #[rr::field("tt")]
    _marker: PhantomData<T>,
}
```

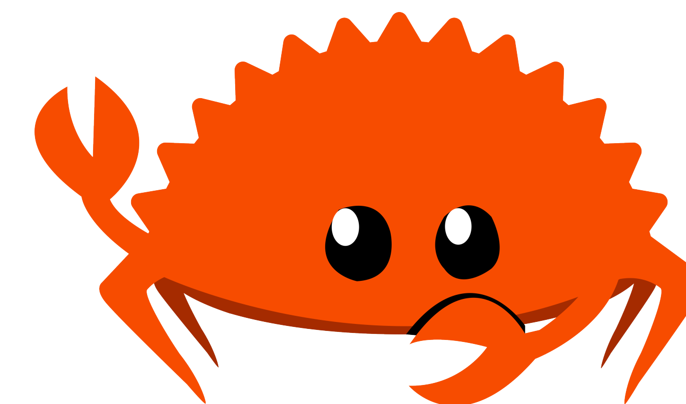
(simplified invariant, does not handle the case that `T` is a ZST)

CUSTOM REPRESENTATION INVARIANTS

`Vec` exposes the list of its initialised elements

```
pub struct Vec<T> {
    #[rr::field("(l, cap)")]
    buf: RawVec<T>,
    #[rr::field("len")]
    len: usize,
}
```





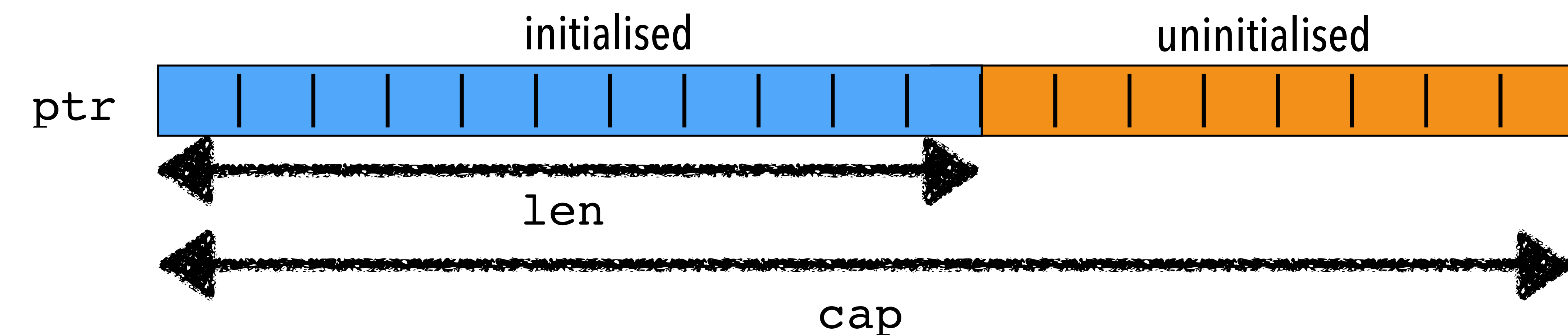
(simplified invariant, does not handle the case that `T` is a ZST)

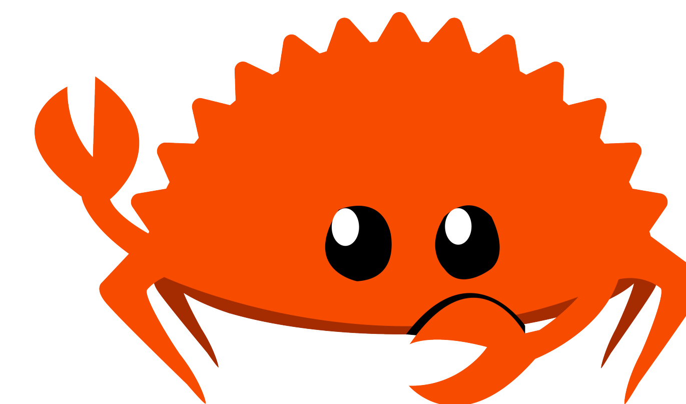
CUSTOM REPRESENTATION INVARIANTS

`Vec` exposes the list of its initialised elements

```
#[rr::refined_by("xs" : "list (bor {rt_of T})")]
```

```
pub struct Vec<T> {
    #[rr::field("(l, cap)")]
    buf: RawVec<T>,
    #[rr::field("len")]
    len: usize,
}
```





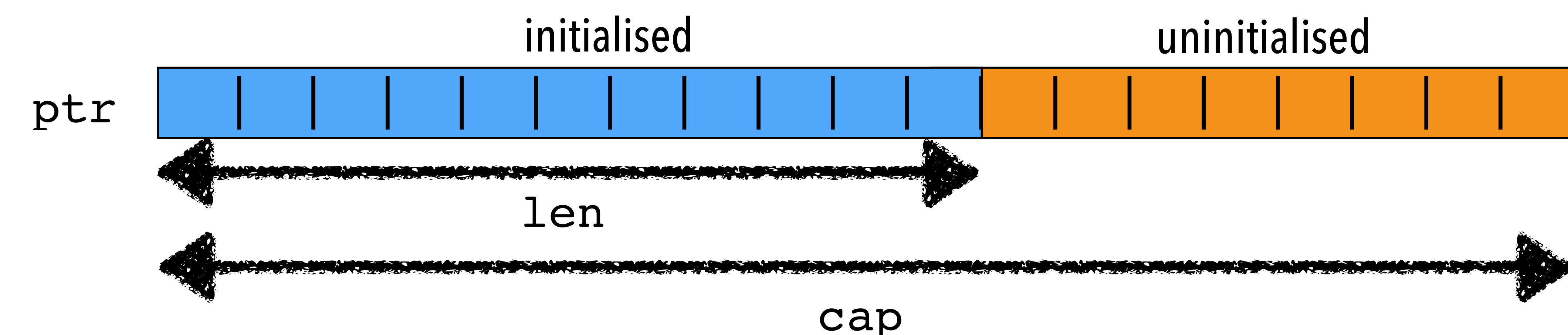
(simplified invariant, does not handle the case that `T` is a ZST)

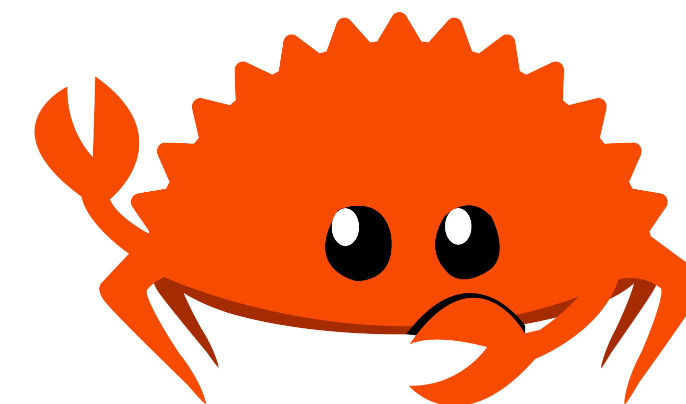
CUSTOM REPRESENTATION INVARIANTS

`Vec` exposes the list of its initialised elements

```
#[rr::refined_by("xs" : "list (bor {rt_of T})")]
#[rr::exists("cap" : "nat", "l" : "loc", "len" : "nat", "els")]
```

```
pub struct Vec<T> {
    #[rr::field("(l, cap)")]
    buf: RawVec<T>,
    #[rr::field("len")]
    len: usize,
}
```





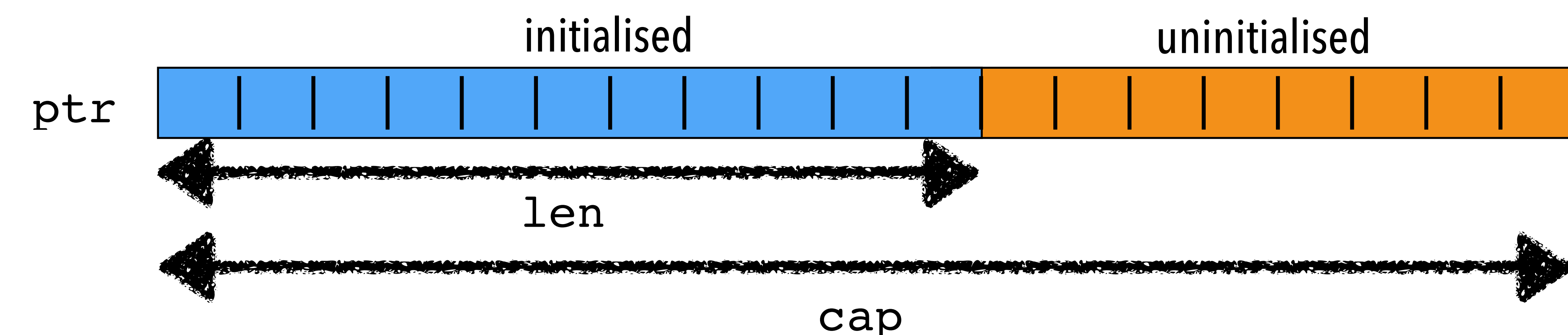
(simplified invariant, does not handle the case that `T` is a ZST)

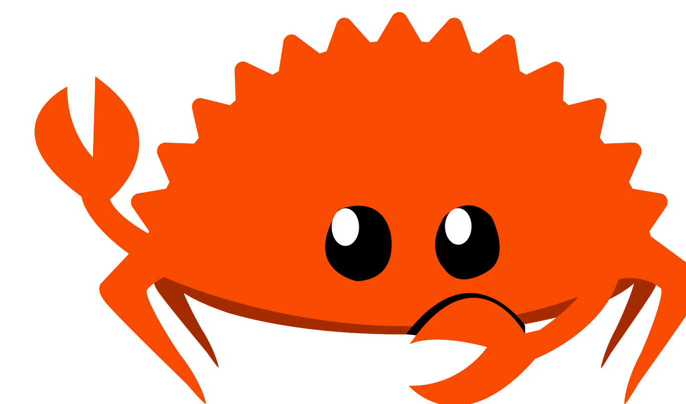
CUSTOM REPRESENTATION INVARIANTS

`Vec` exposes the list of its initialised elements

```
#[rr::refined_by("xs" : "list (bor {rt_of T})")]
#[rr::exists("cap" : "nat", "l" : "loc", "len" : "nat", "els")]
#[rr::invariant(#type "l" : "els" @ "array_t (maybe_init {T}) cap")]
```

```
pub struct Vec<T> {
    #[rr::field("(l, cap)")]
    buf: RawVec<T>,
    #[rr::field("len")]
    len: usize,
}
```





(simplified invariant, does not handle the case that `T` is a ZST)

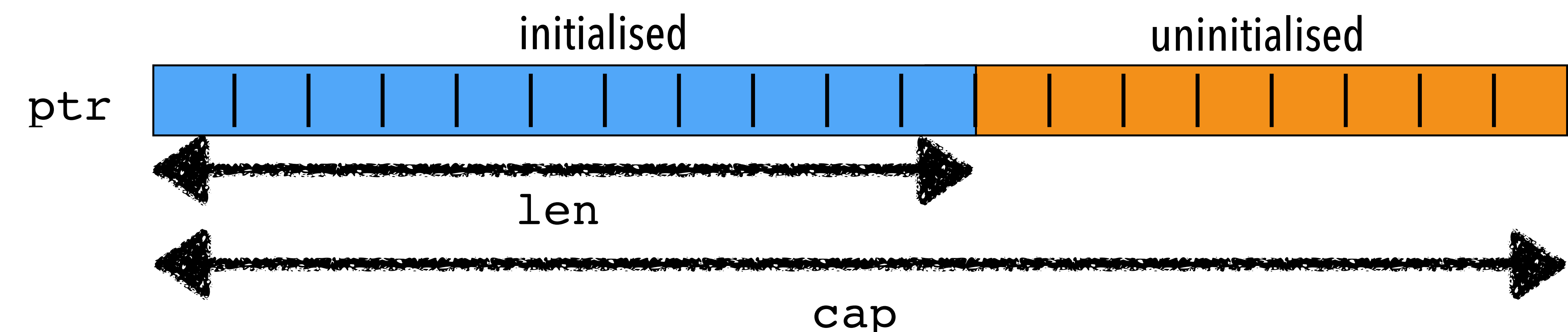
CUSTOM REPRESENTATION INVARIANTS

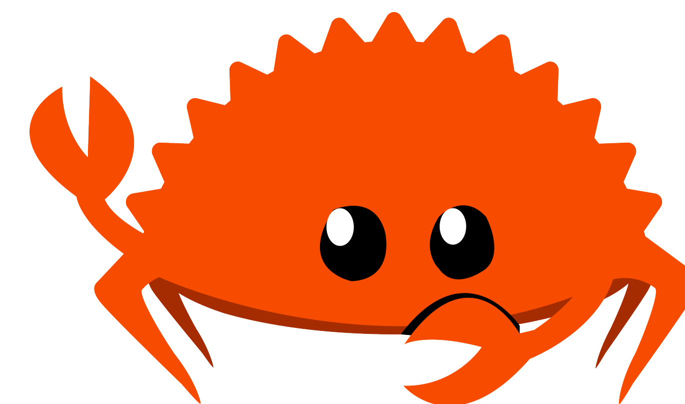
`Vec` exposes the list of its initialised elements

```
#[rr::refined_by("xs" : "list (bor {rt_of T})")]
#[rr::exists("cap" : "nat", "l" : "loc", "len" : "nat", "els")]
#[rr::invariant(#type "l" : "els" @ "array_t (maybe_init {T}) cap")]
#[rr::invariant("⌜∀ i, 0 ≤ i < len →
                   els !!! i = #(Some (xs !!! i))⌝")]
```

(initialised)

```
pub struct Vec<T> {
    #[rr::field("(l, cap)")]
    buf: RawVec<T>,
    #[rr::field("len")]
    len: usize,
}
```





(simplified invariant, does not handle the case that `T` is a ZST)

CUSTOM REPRESENTATION INVARIANTS

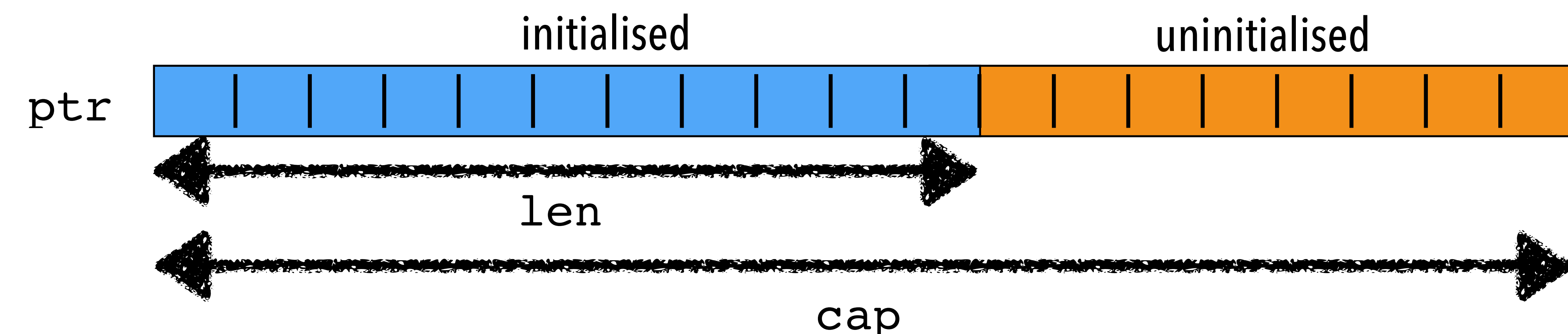
`Vec` exposes the list of its initialised elements

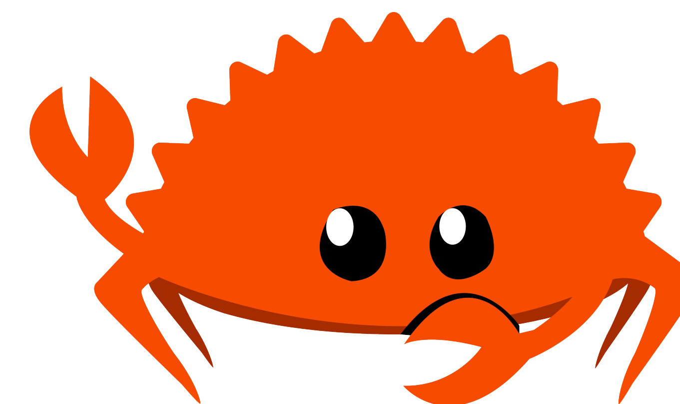
```
#[rr::refined_by("xs" : "list (bor {rt_of T})")]
#[rr::exists("cap" : "nat", "l" : "loc", "len" : "nat", "els")]
#[rr::invariant(#type "l" : "els" @ "array_t (maybe_init {T}) cap")]
#[rr::invariant("⌜∀ i, 0 ≤ i < len →
                 els !!! i = #(Some (xs !!! i))⌝")]
#[rr::invariant("⌜∀ i, len ≤ i < cap → els !!! i = #None⌝")]
```

(initialised)

(uninitialised)

```
pub struct Vec<T> {
    #[rr::field("(l, cap)")]
    buf: RawVec<T>,
    #[rr::field("len")]
    len: usize,
}
```





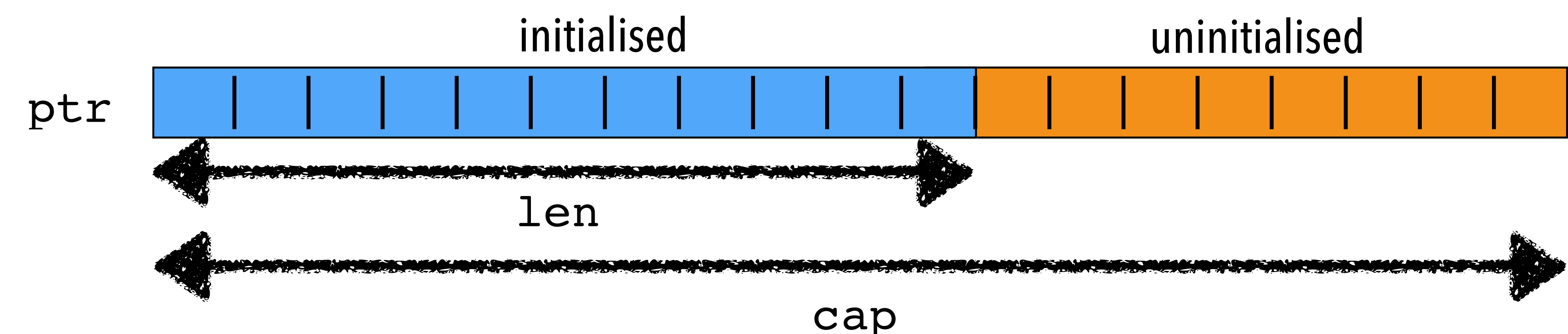
(simplified invariant, does not handle the case that `T` is a ZST)

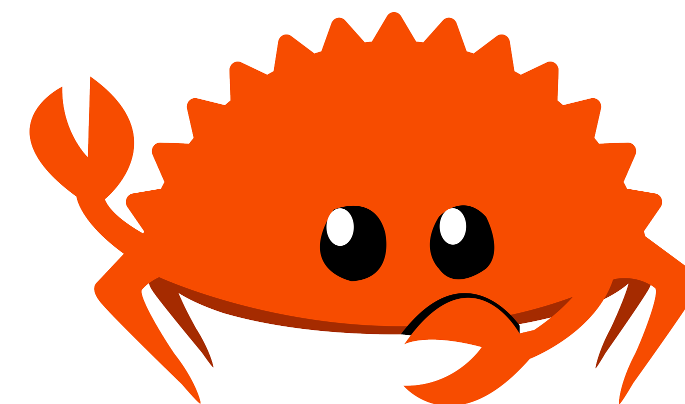
CUSTOM REPRESENTATION INVARIANTS

`Vec` exposes the list of its initialised elements

```
#[rr::refined_by("xs" : "list (bor {rt_of T})")]
#[rr::exists("cap" : "nat", "l" : "loc", "len" : "nat", "els")]
#[rr::invariant(#type "l" : "els" @ "array_t (maybe_init {T}) cap")]
#[rr::invariant("⌜∀ i, 0 ≤ i < len →
                  els !!! i = #(Some (xs !!! i))⌝")]          (initialised)
#[rr::invariant("⌜∀ i, len ≤ i < cap → els !!! i = #None⌝")]  (uninitialised)
#[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, cap)")]
    buf: RawVec<T>,
    #[rr::field("len")]
    len: usize,
}
```





(simplified invariant, does not handle the case that `T` is a ZST)

VERIFYING PUSH

```
pub fn push(&mut self, elem: T) {
    if self.len == self.cap() {
        self.buf.grow();
    }
    unsafe {
        ptr::write(self.ptr().add(self.len), elem);
        // Can't overflow, we'll OOM first.
        self.len += 1;
    }
}
```

VERIFYING PUSH

```
#[rr::params("xs", "γ", "x")]
#[rr::args("(#xs, γ)", "x")]


#[rr::ensures("Obs γ (xs ++ [ #x])")]
pub fn push(&mut self, elem: T) {
    if self.len == self.cap() {
        self.buf.grow();
    }
    unsafe {
        ptr::write(self.ptr().add(self.len), elem);
        // Can't overflow, we'll OOM first.
        self.len += 1;
    }
}
```

VERIFYING PUSH

```
#[rr::params("xs", "γ", "x")]
#[rr::args("(#xs, γ)", "x")]


#[rr::ensures("Obs γ (xs ++ [ #x])")]
pub fn push(&mut self, elem: T) {
    if self.len == self.cap() {
        self.buf.grow();
    }
    unsafe {
        ptr::write(self.ptr().add(self.len), elem);
        // Can't overflow, we'll OOM first.
        self.len += 1;
    }
}
```

VERIFYING PUSH

```
#[rr::params("xs", "γ", "x")]
#[rr::args("(#xs, γ)", "x")]
#[rr::requires("⌜length xs < max_int usize_t⌝")]
#[rr::requires("⌜size_of_array_in_bytes {st_of T}
                (2 * length xs) ≤ max_int isize_t⌝")]
#[rr::ensures("Obs γ (xs ++ [ #x])")]
pub fn push(&mut self, elem: T) {
    if self.len == self.cap() {
        self.buf.grow();
    }
    unsafe {
        ptr::write(self.ptr().add(self.len), elem);
        // Can't overflow, we'll OOM first.
        self.len += 1;
    }
}
```

VERIFYING PUSH

```
#[rr::params("xs", "γ", "x")]
#[rr::args("(#xs, γ)", "x")]
#[rr::requires("⌜length xs < max_int usize_t⌝")]
#[rr::requires("⌜size_of_array_in_bytes {st_of T}
                  (2 * length xs) ≤ max_int isize_t⌝")]
#[rr::ensures("Obs γ (xs ++ [ #x])")]
pub fn push(&mut self, elem: T) {
    if self.len == self.cap() {
        self.buf.grow();
    }
    unsafe {
        ptr::write(self.ptr().add(self.len), elem);
        // Can't overflow, we'll OOM first.
        self.len += 1;
    }
}
```

Representation invariant of `Vec` broken (ownership moved to `ptr::write`)!

CONCLUSIONS

EVALUATION

	Rust LOC	Sideconds	Spec + manual proof	Verification time
Vec::new	6	20	1 + 0	20s
Vec::push	9	128	5 + 35	7min 25s
Vec::pop	8	79	4 + 17	3min 30s
Vec::get_unchecked_mut	7	47	5 + 10	2min
Vec::get_mut	8	73	7 + 40	1min 05s
Vec::get_unchecked	7	31	4 + 0	55s
Vec::get	8	56	4 + 0	1min 20s
Total (RawVec + Vec)	**120 (14 functions)**	**400 ++**	**75 + 105**	**8.5min (wall)**

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

EVALUATION

	Rust LOC	Sideconds	Spec + manual proof	Verification time
Vec::new	6	20	1 + 0	20s
Vec::push	9	128	5 + 35	7min 25s
Vec::pop	8	79	4 + 17	3min 30s
Vec::get_unchecked_mut	7	47	5 + 10	2min
Vec::get_mut	8	73	7 + 40	1min 05s
Vec::get_unchecked	7	31	4 + 0	55s
Vec::get	8	56	4 + 0	1min 20s
Total (RawVec + Vec)	**120 (14 functions)**	**400 ++**	**75 + 105**	**8.5min (wall)**

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

Support libraries: `mem::{size_of, align_log_of, align_of}, Box::new,`

```
alloc::{alloc, dealloc, realloc},
ptr::{write, read, invalid, copy_nonoverlapping},
mut_ptr::{add, offset}
```


L. Gäher, M. Sammler, R. Jung, R. Krebbers, D. Dreyer

TAKEAWAYS FOR FOUNDATIONAL VERIFICATION

TAKEAWAYS FOR FOUNDATIONAL VERIFICATION

foundational verification is expensive

- small Rust code (120 lines) → huge MIR (900 lines)

TAKEAWAYS FOR FOUNDATIONAL VERIFICATION

foundational verification is expensive

- small Rust code (120 lines) → huge MIR (900 lines)

foundational verification allows to build intricate but sound type systems

- would be very hard to get right without mechanised proofs

TAKEAWAYS FOR FOUNDATIONAL VERIFICATION

foundational verification is expensive

- small Rust code (120 lines) → huge MIR (900 lines)

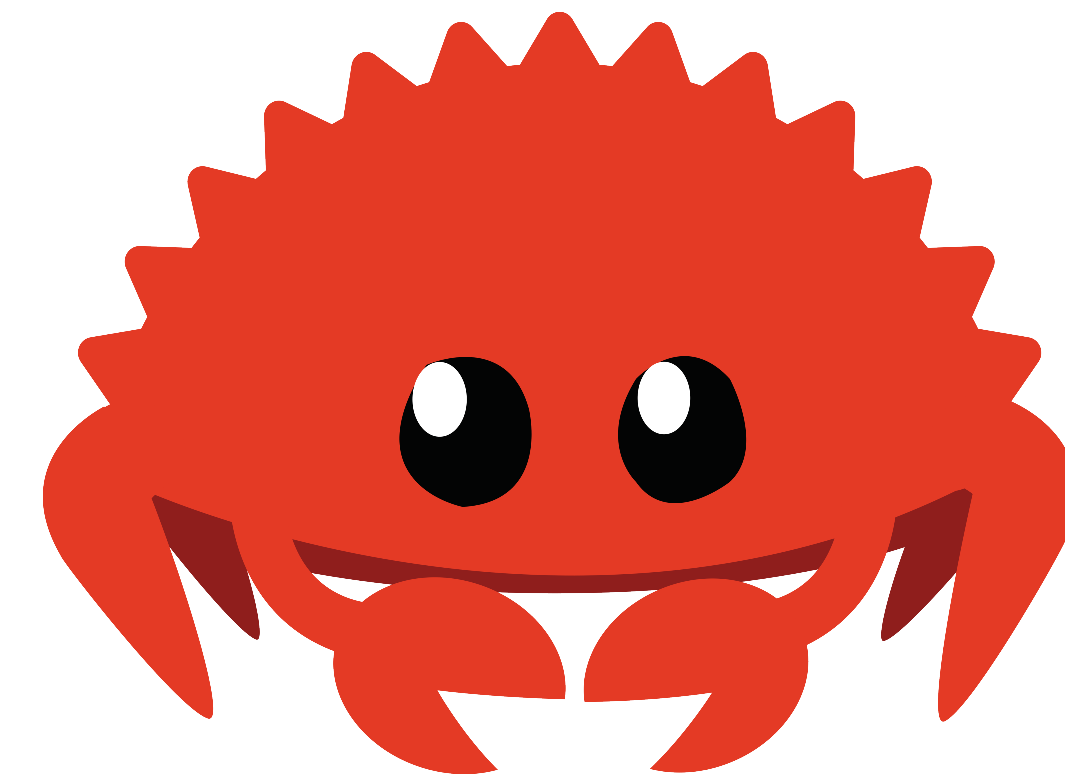
foundational verification allows to build intricate but sound type systems

- would be very hard to get right without mechanised proofs

foundational verification can be automated

- but is not yet at the level of non-foundational tools

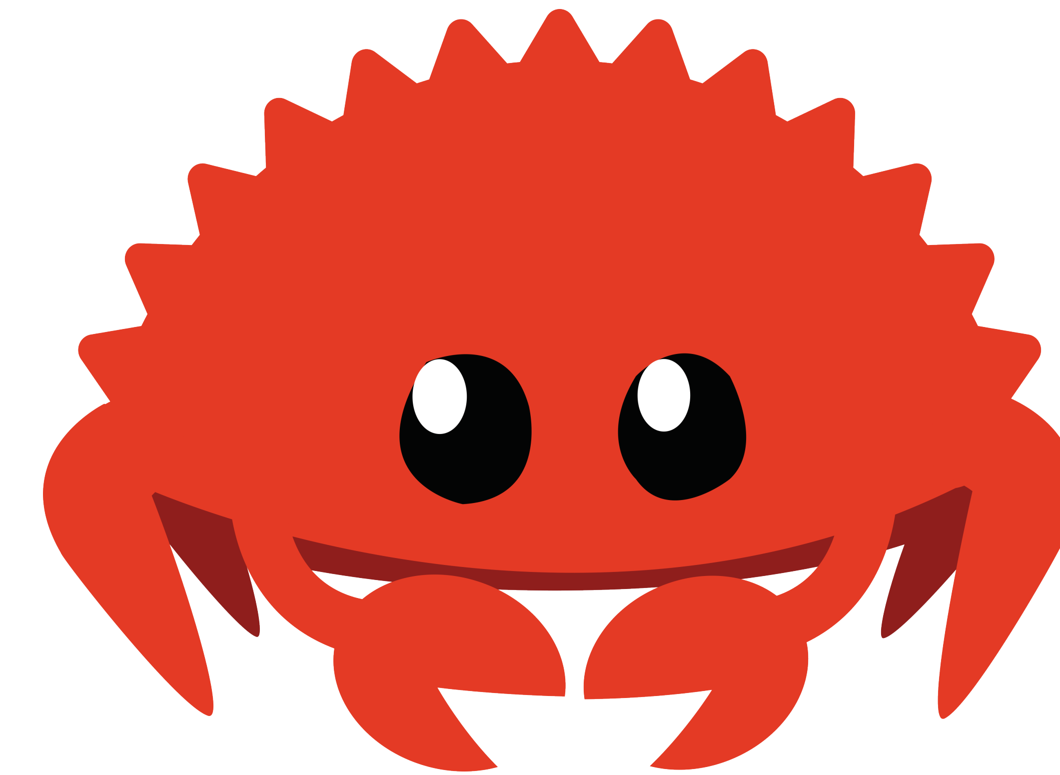
TAKEAWAYS FOR VERIFYING UNSAFE CODE



TAKEAWAYS FOR VERIFYING UNSAFE CODE

Requires a low-level semantics

- ➔ opposed to high-level semantics sufficient for safe Rust
- ➔ memory model, value representation, …
- ➔ low-level byte model absolutely crucial for things like `copy_nonoverlapping`



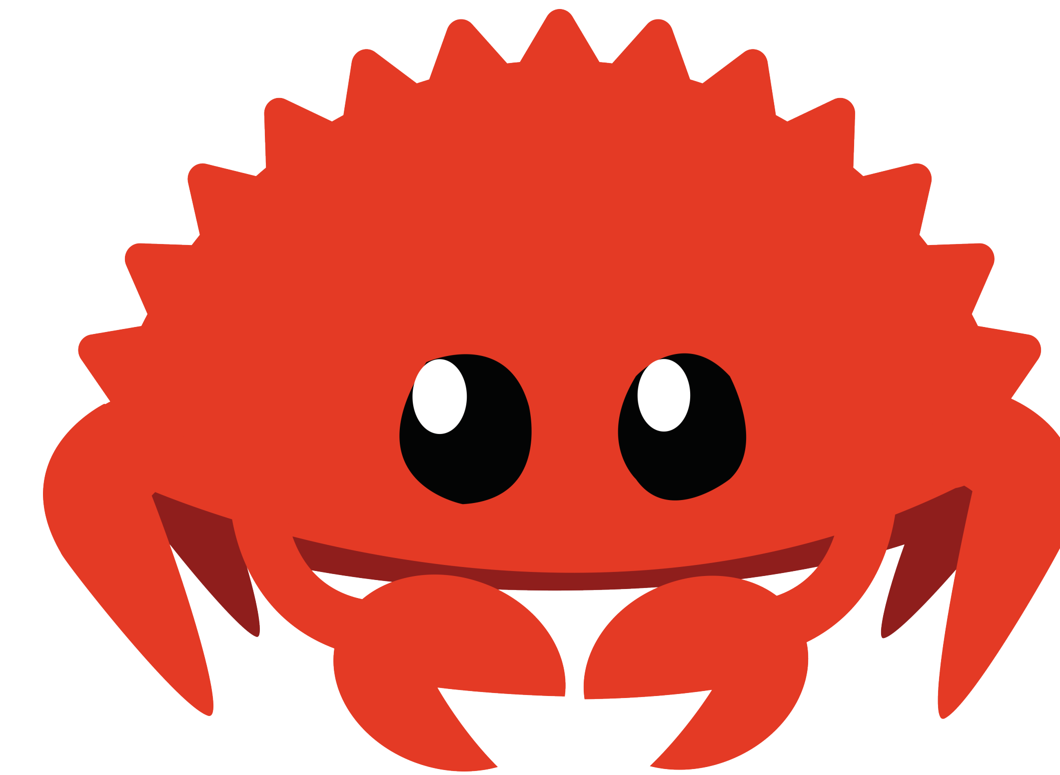
TAKEAWAYS FOR VERIFYING UNSAFE CODE

Requires a low-level semantics

➔ opposed to high-level semantics sufficient for safe Rust

➔ memory model, value representation, …

➔ low-level byte model absolutely crucial for things like `copy_nonoverlapping`



ZSTs are a special pain to deal with

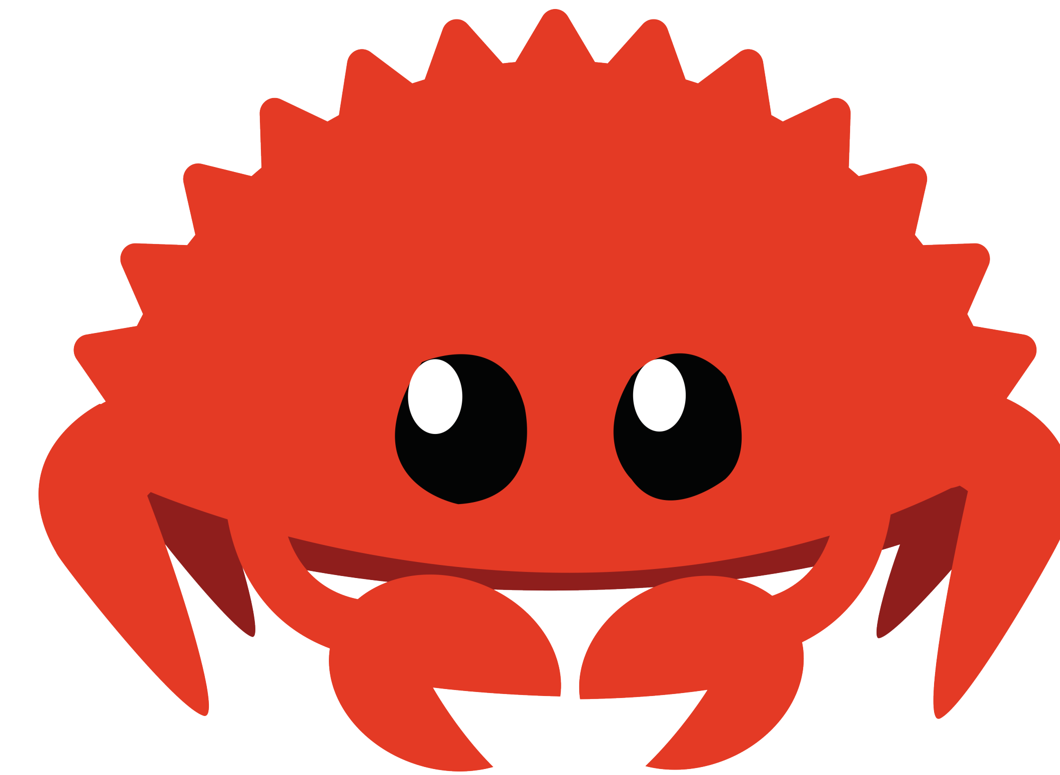
➔ regular surprises by unexpected corner cases

➔ e.g. pointer arithmetic rules

TAKEAWAYS FOR VERIFYING UNSAFE CODE

Requires a low-level semantics

- ➔ opposed to high-level semantics sufficient for safe Rust
- ➔ memory model, value representation, …
- ➔ low-level byte model absolutely crucial for things like `copy_nonoverlapping`



ZSTs are a special pain to deal with

- ➔ regular surprises by unexpected corner cases
- ➔ e.g. pointer arithmetic rules

Interaction with references is tricky

- ➔ in safe code, have (essentially functional) abstractions
- ➔ in unsafe code, need to access low-level representation
- ➔ having a RustBelt-style model helps to seamlessly integrate

REFINEDRUST

An **automatic translation** scheme from Rust into Radium

Proof automation based on RefinedC's Lithium engine

Radium **operational semantics** for Rust based on RefinedC

Refinement type system with semantic model inspired by RustBelt

✓ **Lift limitations of RustBelt**
- allow automatic translation from Rust
- handle more borrow patterns
- develop a new notion of place types
- extend RustBelt's lifetime logic

✓ **Equip RefinedC with references**
- redesigned RefinedC's refinement model