Scalable Bug Detection for Internally Unsafe Libraries

A Logical Approach to Type Refutation

Pedro Carrott, Sacha-Élie Ayoun, Azalea Raad
Imperial College London

TPSA 2025

Motivation: Unsafe Rust

The Rust programming language is seen as a safe alternative to C/C++.

Rust attempts to combine the benefits of both high-level safety and low-level control.

Unsafe escape hatch: write safe abstractions around unsafe code.
(when the type system is too strong)

Safe Abstractions of Unsafe Code

A safe abstraction of even numbers in Rust:

```
fn zero() → Even {
     Even { val: 0 }
}

unsafe fn succ(x : Even) → Even {
     Even { val: x.val + 1 }
}

fn next(x : Even) → Even {
     unsafe { succ(succ(x)) }
}
```

Example taken from RefinedRust

A function that relies on the Even abstraction:

```
fn naive(x : Even) → () {
     if (x.val % 2 != 0) {
          // Cannot be reached
          // without unsafe blocks
          unsafe { UB() }
     }
}
```

Safe Abstractions of Unsafe Code

A safe abstraction of even numbers in Rust:

```
fn zero() → Even {
    Even { val: 0 }
}

unsafe fn succ(x : Even) → Even {
    Even { val: x.val + 2 }
}

fn next(x : Even) → Even {
    unsafe { succ(succ(x)) }
}
```

A function that relies on the Even abstraction:

```
→ () {
!= 0) {
be reached
t unsafe blocks
UB() }
```

Type signatures in Rust encode specifications

Example taken from RefinedRust

Safe Abstractions of Unsafe Code

An unsafe abstraction of even numbers in Rust:

```
fn zero() → Even {
     Even { val: 0 }
}

fn succ(x : Even) → Even {
     Even { val: x.val + 1 }
}

fn next(x : Even) → Even {
     succ(succ(x))
}
```

A function that relies on the Even abstraction:

```
fn naive(x : Even) → () {
     if (x.val % 2 != 0) {
          // May be reached
          // without unsafe blocks
          unsafe { UB() }
     }
}
```

Safe Abstractions of Unsafe Code

An (unsafe) abstraction of even numbers in C:

```
Even zero() {
    return (Even){.val = 0 };
}

Even succ(Even x) {
    return (Even){.val = x.val + 1};
}

Even next(Even x) {
    return succ(succ(x));
}
```

A function that relies on the **Even** abstraction:

```
void naive(Even x) {
    if (x.val % 2 != 0) {
        // May be reached
        // if called from main
        UB()
    }
}
```

Type Soundness in Rust

Safe functions in Rust
No undefined behaviour for well-typed inputs.

Functions in C
Undefined behaviour is always possible.

*Static analysis tools can **automatically** report type unsoundnesses in Rust, without requiring an executable path (as in C).*

(in Rust, the notion of manifest error is much larger)

Reasoning about Internal Unsafety

Verification. *Known solutions:*

- RustBelt (foundational)
- Gillian-Rust (automated)
- RefinedRust (both)

Correctness: over-approximate (OX) specifications via Hoare logic.

Type invariants: user-provided semantic interpretation of types.

Refutation. *No known solutions.*

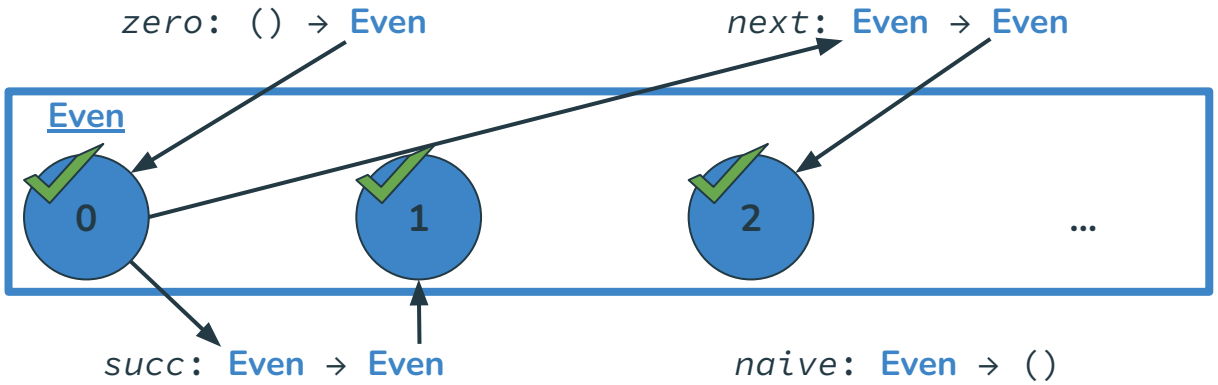
- Our approach: RUXt
- Detection of type unsoundness
- Fully automated, no annotations

Incorrectness: under-approximate (UX) specifications via incorrectness logic.

Type spaces: inference of well-typed terms via symbolic execution.

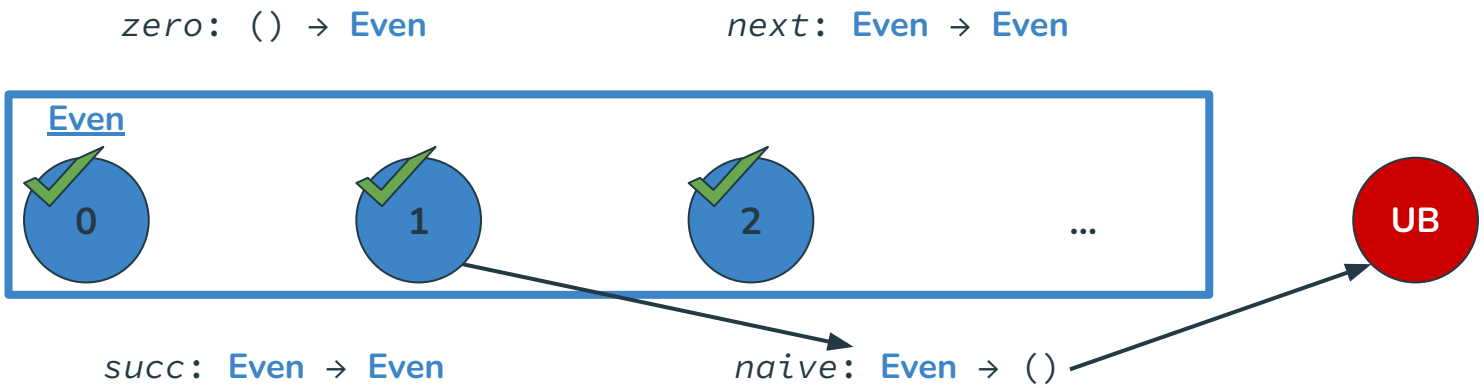
Under-Approximate Type Spaces

Symbolic execution of safe functions for well-typed inputs results in well-typed terms.



Under-Approximate Type Spaces

Undefined behaviour refutes safety: the library does not safely encapsulate its unsafe code.



Under-Approximate Type Spaces

We can construct a program with only safe calls to the library that results in undefined behaviour:

```
let x = zero();   // x is 0
let y = next(x);  // y is 2
let z = succ(y);  // z is 3
naive(z)          // UB
```

```
let x = zero();   // x is 0
let y = succ(x);  // y is 1
let z = next(y);  // z is 3
naive(z)          // UB
```

We explore the space of types, not all possible traces.

The Type Refutation Algorithm

1. **Σ** = ∅ `// Inferred type spaces`
2. Pick (safe) function *f* and inputs from **Σ**.
3. Symbolically execute *f*:
   a. If UB ➤ **Refuted safety!**
   b. Otherwise, update **Σ** with return state.
   Repeat from 2.

False Negatives: the algorithm may fail to detect provably unsafe abstractions.

***NO** False Positives*: every refuted type is a provably unsafe abstraction.

Contributions

- RUXt: prototype OCaml implementation for λ_{RUXt} (inspired by RustBelt's λ_{Rust}).
  No references, work in progress!

- Full Rocq formalisation: λ_{RUXt} semantics and UX/OX logical framework.

- Partial Rocq formalisation: soundness (in progress!)