ECOOP '25 Bergen

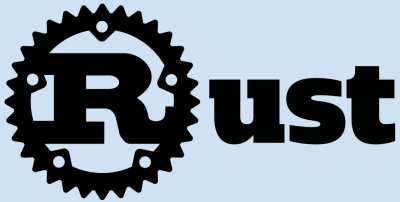
Compositional Bug Detection for Internally Unsafe Libraries

A Logical Approach to Type Unsoundness

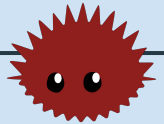
Pedro Carrott [1] Sacha-Élie Ayoun [1] Azalea Raad [1]

[1] *Imperial College London*

Safe Abstractions of Unsafe Code

Our motivation: 

Rust is a safe alternative to C/C++ ➡ *high-level safety* ✚ *low-level control*



Unsafe escape hatch: write safe abstractions around unsafe code

(when the type system is too strong)

Safe Abstractions of Unsafe Code

The Even type: an abstraction of even numbers

A safe abstraction 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)) }
}
```

An internally unsafe function

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

Example taken from RefinedRust

Safe Abstractions of Unsafe Code

The Even type: an abstraction of even numbers

A safe abstraction 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)) }
}
```

An internally unsafe function

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

Type safety

Type signatures in Rust encode specifications

Example taken from RefinedRust

Safe Abstractions of Unsafe Code

The Even type: an abstraction of even numbers

An unsafe abstraction 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))
}
```

An internally unsafe function

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

Safe Abstractions of Unsafe Code

The Even type: an abstraction of even numbers

An unsafe abstraction in Rust

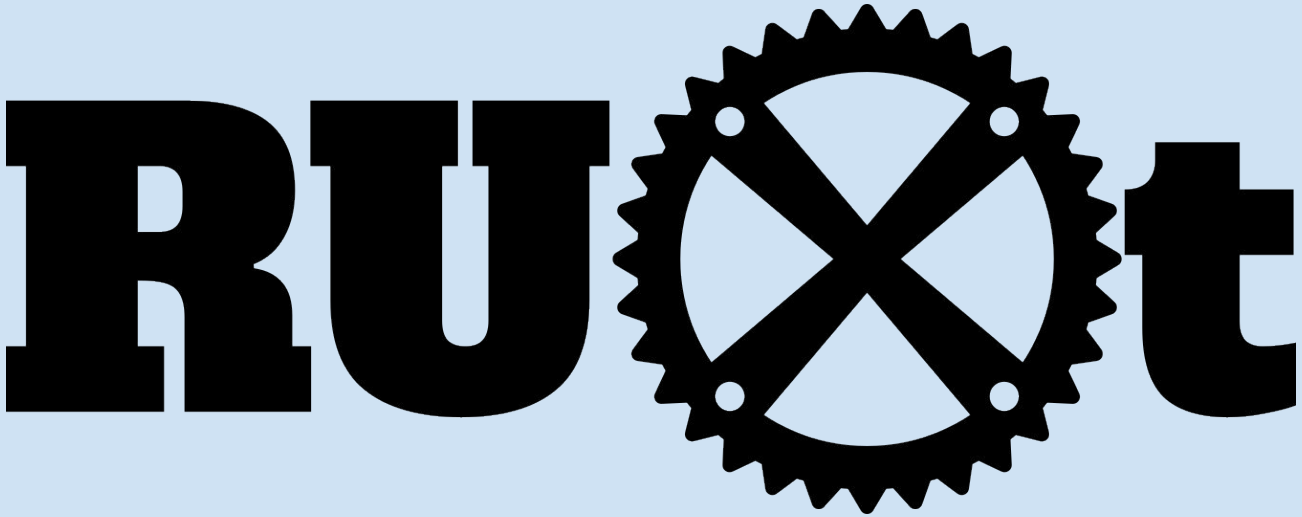
```
fn zero() → Even {
    Even { val: 0 }
}

fn succ(x : Even) → E
    Even { val: x.va
}

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

An internally unsafe function

```
() {
= 0 {
reached
unsafe blocks
UB() }
    }
}
```

Type safety violation

succ **and** *noop* **cannot co-exist as safe functions**



A Compositional Analysis for Automatic Type Safety Refutation

Reasoning about Unsafe Code

Verification. *Known solutions:*

- RustBelt (foundational)
- Gillian-Rust, Verus (semi-automated)
- RefinedRust (both)

Refutation. *No known solutions.*

- Our approach: **RUXt**
- Detection of type safety violations
- Fully automated, no annotations

Correctness: over-approximate specifications via Hoare logic.

Type invariants: superset of safe values, user-defined for each type.

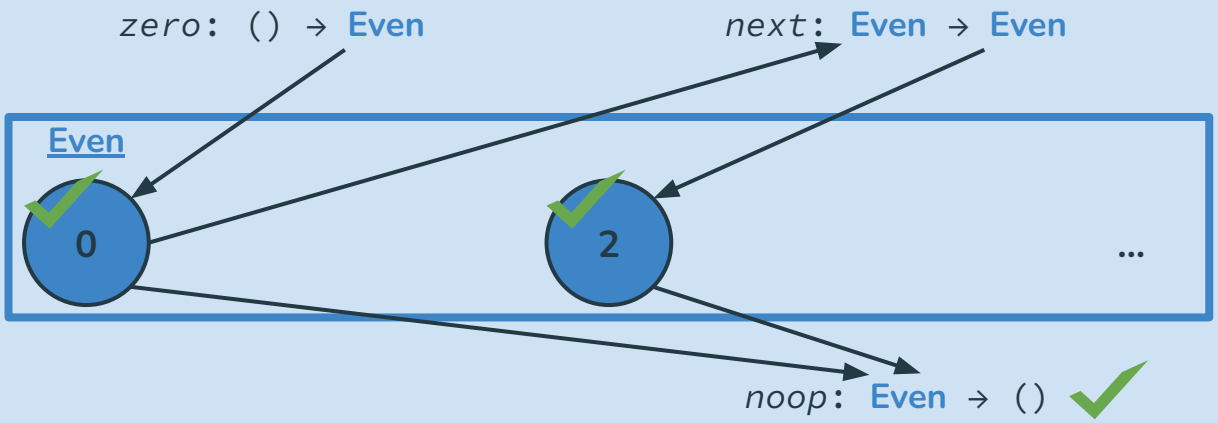
Incorrectness: under-approximate specifications via incorrectness logic.

Type subvariants: subset of safe values, inferred via symbolic execution.

The RUXt Algorithm

Type Space Inference

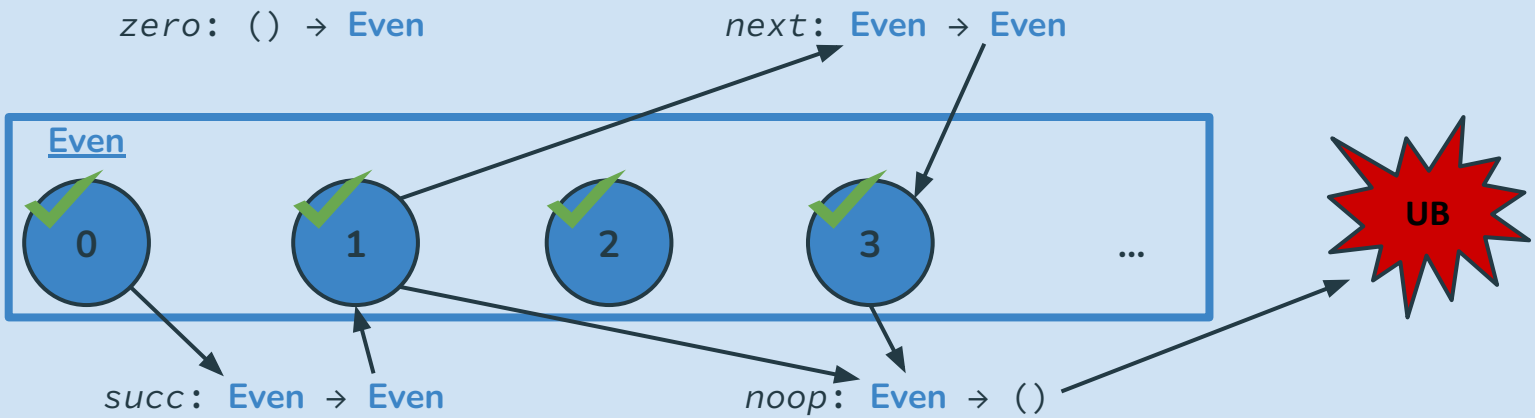
Symbolic execution Safe functions for well-typed inputs result in safe values



The RUXt Algorithm

Type Safety Refutation

Undefined behaviour The library does not encapsulate its unsafe code if it yields UB



The RUXt 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 and 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

Type Space Exploration

We can construct programs with only safe calls to the library that result in undefined behaviour

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

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

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

Handling References

For updates via references, we implement *reference-free wrappers*

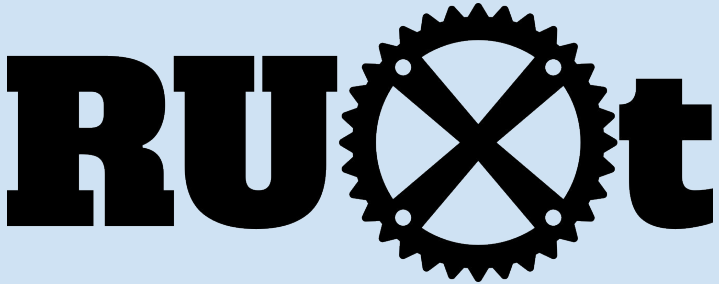
```
fn next(&mut Even) → ()
```

```
fn next_wrap(mut x: Even) → Even {
    next(&mut x); x
}
```

```
fn f(&mut T) → &mut U
```

```
fn f_wrap(mut x: T, y: U) → T {
    *(f(&mut x)) = y; x
}
```

Contributions



Algorithm

- *Compositional* and *fully automated* analysis for type safety refutation
- Reference handling via reference-free wrappers

Formalisation

- Rocq *mechanisation* for λ_{RUXt} (inspired by λ_{Rust} from RustBelt)
- True-positives result: *inadequacy theorem*

Implementation

- Prototype in OCaml for bug detection in simple Rust libraries

Future Work

Algorithm
vs
Implementation

- ➢ Automate wrapper generation
- ➢ Extract witness programs

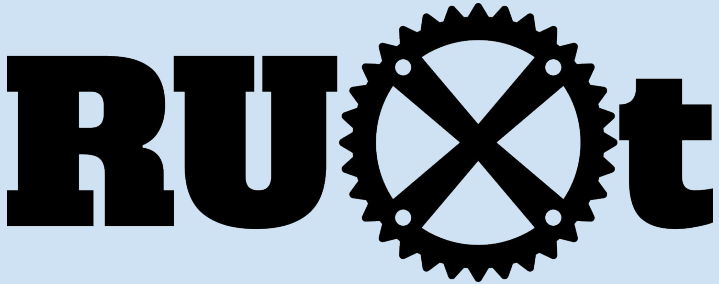
Yet-unexplored Rust features

Polymorphism Traits Higher-order functions Concurrency Stacked/Tree Borrows

Main Goal

Large scale evaluation in real Rust code

Contributions



Algorithm

- *Compositional* and *fully automated* analysis for type safety refutation
- Reference handling via reference-free wrappers

Formalisation

- Rocq *mechanisation* for λ_{RUXt} (inspired by λ_{Rust} from RustBelt)
- True-positives result: *inadequacy theorem*

Implementation

- Prototype in OCaml for bug detection in simple Rust libraries

Thank you!