THRUST: A Prophecy-Based Refinement Type System for RUST

PLDI 2025

Hiromi Ogawa
University of Tsukuba

Taro Sekiyama
National Institute of Informatics

Hiroshi Unno
Tohoku University

Rust: General-Purpose Programming Language

- Adopted across a wide range of software development fields
  - from Linux kernel[1] to world's largest package registry[2]
- Allows low-level operations for high-performance applications and system programming while ensuring memory safety through its unique type system
- Also emphasizes developer productivity through its language features and tooling

[1] https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=8aebac82933ff1a7c8eede18cab11e1115e2062b

[2] https://www.rust-lang.org/static/pdfs/Rust-npm-Whitepaper.pdf

Two Categories of Rust Code: **Safe** and **Unsafe**

- In normal (safe) Rust, **the type system ensures** memory-safety
  - Pointer manipulation is constrained
  - This constraint provides many good properties for verification
- In unsafe Rust, **users are responsible for ensuring** memory safety
  - Raw pointer operations are permitted
  - Static verification is harder

Two Categories of RUST Code: **Safe** and **Unsafe**

- In normal (safe) RUST, **the type system ensures** memory-safety
  - Pointer manipulation is constrained
  - This constraint provides many good properties for verification
- In unsafe RUST, **users are responsible for ensuring** memory safety
  - Raw pointer operations are permitted
  - Static verification is harder

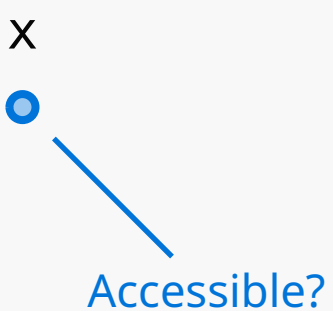
Our work (THRUST) focuses on safe Rust to leverage safe Rust guarantees for higher-level static verification

Good Property of Safe Rust: Aliasing XOR Mutability

- **Aliasing** means a situation where multiple pointers are sharing the same memory location
- In (safe) Rust, one of the following conditions must apply to pointers:
  - There are **multiple** aliases; all of them are **immutable** (read-only)
  - There is **only one** accessible alias; it may be **mutable** (writable)
- This property is often called "**Aliasing XOR Mutability**"

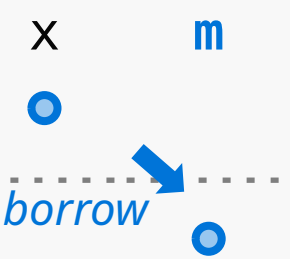
Good Property of Safe Rust: Aliasing XOR Mutability

```
let mut x = Box::new(2);
```



Good Property of Safe Rust: Aliasing XOR Mutability

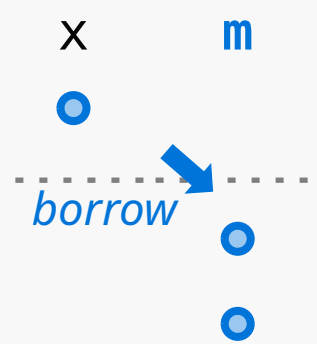
```
let mut x = Box::new(2);
.....................................
let m = &mut *x;
```



x cannot be used

Good Property of Safe Rust: Aliasing XOR Mutability

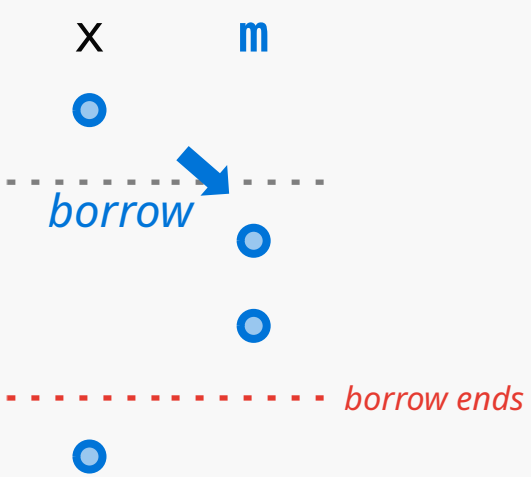
```
let mut x = Box::new(2);
let m = &mut *x;
*m = 1;
```



`x` cannot be used / Mutation via `m` is allowed

Good Property of Safe Rust: Aliasing XOR Mutability

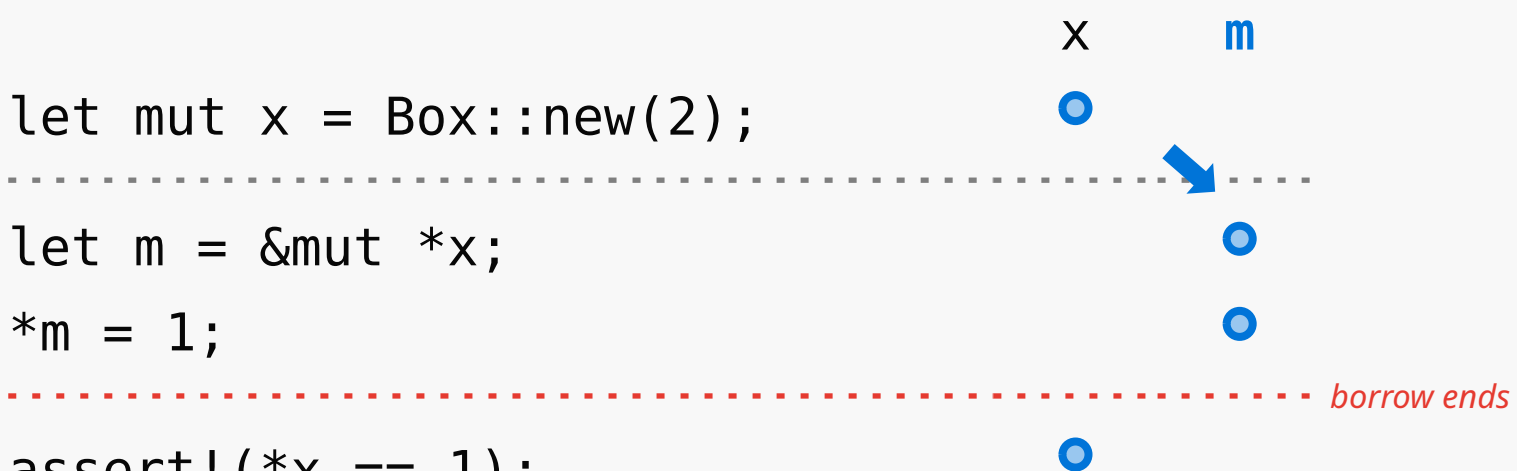
```
let mut x = Box::new(2);
let m = &mut *x;
*m = 1;
assert!(*x == 1);
```



x can be used again

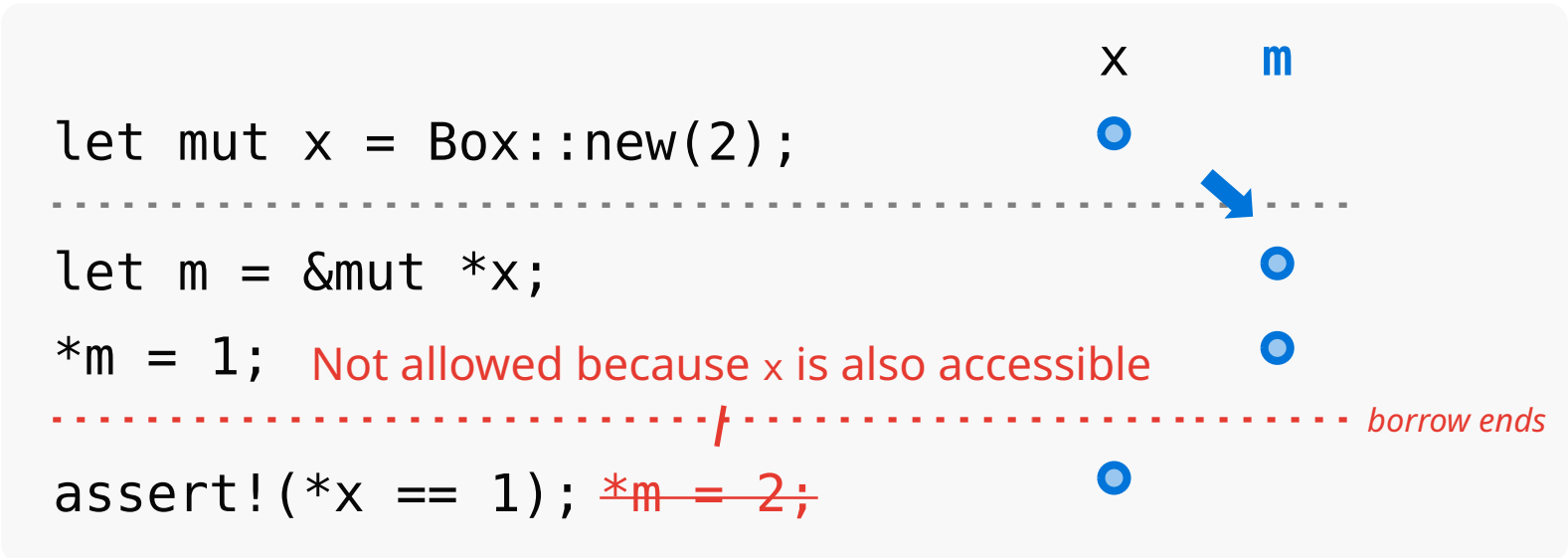
Good Property of Safe Rust: Aliasing XOR Mutability

```
let mut x = Box::new(2);
let m = &mut *x;
*m = 1;
assert!(*x == 1);
```



The pointer is mutable ⟹ it's only one accessible alias
(aliasing XOR mutability)

Good Property of Safe Rust: Aliasing XOR Mutability



The pointer is mutable ⟹ it's only one accessible alias

(aliasing XOR mutability)

Static Verification Tools for Safe Rust

- “Aliasing XOR mutability” guarantee of safe Rust aids higher-level static verification of functional correctness
- Various static verification tools have been proposed leveraging this property…

Static Verification Tools for Safe Rust

	Automated?	Modular?	Precise?	Datatypes?	Higher-Order?
RustHorn Matsushita et al. (2021)	●	○	●	◐	○
Creusot Denis et al. (2022)	○	●	●	●	●
Flux Lehmann et al. (2023)	◐	●	◐	●	○
RefinedRust Gäher et al. (2024)	○	●	●	●	○
Prusti Astrauskas et al. (2019)	○	●	◐	●	○
⋮			⋮		

● yes ○ no ◐ partial yes

Excerpt from Table 1 in the paper

Inductive invariants are inferred

ation Tools for Safe Rust

	Automated?	Modular?	Precise?	Datatypes?	Higher-Order?
RustHorn Matsushita et al. (2021)	●	○	●	◐	○
Creusot Denis et al. (2022)	○	●	●	●	●
Flux Lehmann et al. (2023)	◐	●	◐	●	○
RefinedRust Gäher et al. (2024)	○	●	●	●	○
Prusti Astrauskas et al. (2019)	○	●	◐	●	○
⋮			⋮		

● yes ○ no ◐ partial yes

Excerpt from Table 1 in the paper

Inductive invariants are inferred

Divide verification into smaller parts

at Safe Rust

	Automated?	Modular?	Precise?	Datatypes?	Higher-Order?
RustHorn Matsushita et al. (2021)	●	○	●	◐	○
Creusot Denis et al. (2022)	○	●	●	●	●
Flux Lehmann et al. (2023)	◐	●	◐	●	○
RefinedRust Gäher et al. (2024)	○	●	●	●	○
Prusti Astrauskas et al. (2019)	○	●	◐	●	○
⋮			⋮		

● yes ○ no ◐ partial yes

Excerpt from Table 1 in the paper

Inductive invariants are inferred

Divide verification into smaller parts

Track state changes via pointers precisely

	Automated?	Modular?	Precise?	Datatypes?	Higher-Order?
RustHorn Matsushita et al. (2021)	●	○	●	◐	○
Creusot Denis et al. (2022)	○	●	●	●	●
Flux Lehmann et al. (2023)	◐	●	◐	●	○
RefinedRust Gäher et al. (2024)	○	●	●	●	○
Prusti Astrauskas et al. (2019)	○	●	◐	●	○
⋮			⋮		

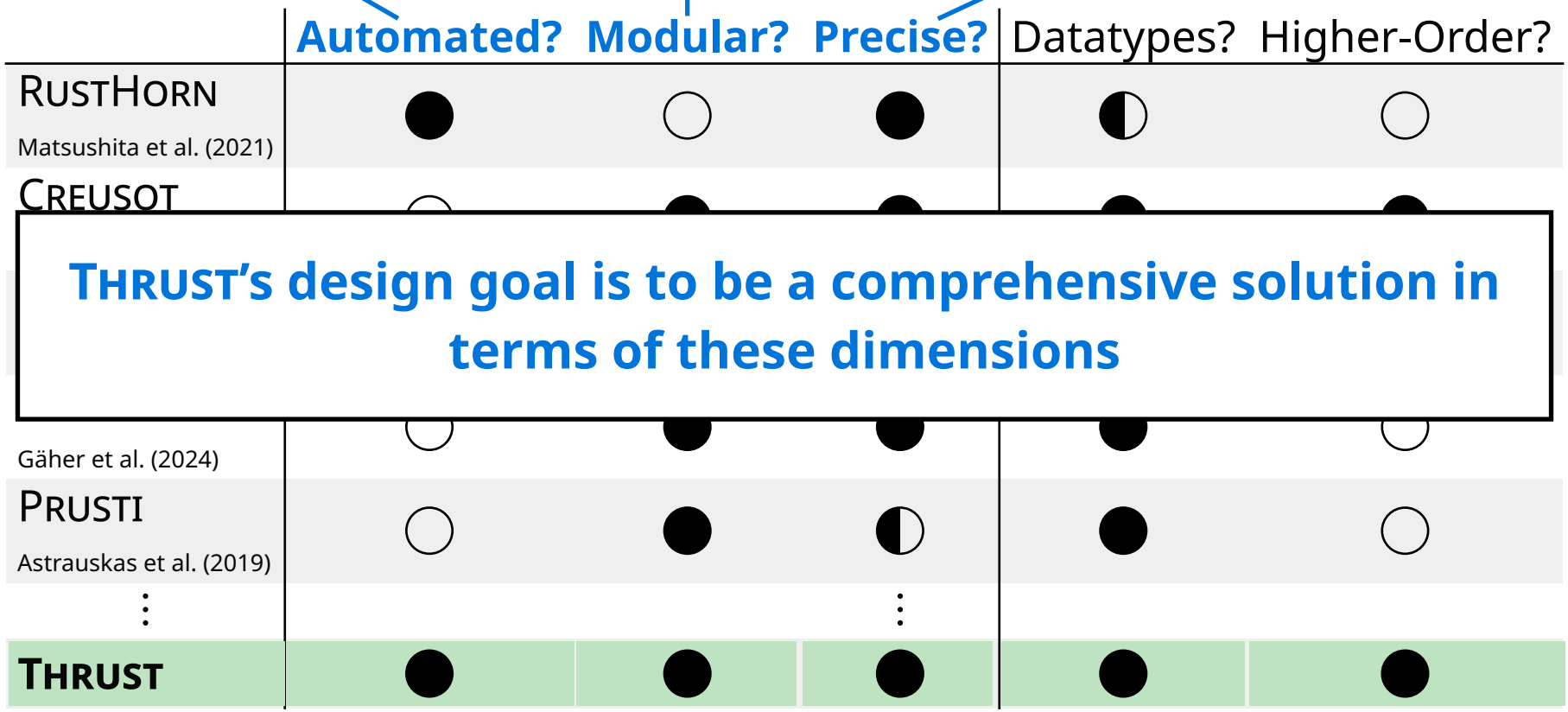
● yes ○ no ◐ partial yes

Excerpt from Table 1 in the paper

Inductive invariants are inferred

Divide verification into smaller parts

Track state changes via pointers precisely



Thrust's design goal is to be a comprehensive solution in terms of these dimensions

● yes ○ no ◐ partial yes

Excerpt from Table 1 in the paper

Contributions

1. We integrated (and the integration poses a technical challenge): Refinement Types + Flow-Sensitivity + Prophecies + CHC Solving

2. We designed a novel refinement type system and proved its soundness under the assumption of “aliasing XOR mutability”

3. We implemented and evaluated the system against other notable Rust verification tools, obtaining promising results

C

Modularity +
Language Features

1. We integrated (and the integration poses a technical challenge): **Refinement Types** + Flow-Sensitivity + Prophecies + CHC Solving

2. We designed a novel refinement type system and proved its soundness under the assumption of “aliasing XOR mutability”

3. We implemented and evaluated the system against other notable Rust verification tools, obtaining promising results

C

Modularity + Language Features

Precision

1. We integrated (and the integration poses a technical challenge): **Refinement Types** + **Flow-Sensitivity + Prophecies** + CHC Solving

2. We designed a novel refinement type system and proved its soundness under the assumption of “aliasing XOR mutability”

3. We implemented and evaluated the system against other notable Rust verification tools, obtaining promising results

C

Modularity + Language Features

Precision

Automation

1. We integrated (and the integration poses a technical challenge): **Refinement Types** + **Flow-Sensitivity + Prophecies** + **CHC Solving**

2. We designed a novel refinement type system and proved its soundness under the assumption of “aliasing XOR mutability”

3. We implemented and evaluated the system against other notable Rust verification tools, obtaining promising results

Refinement Types Freeman and Pfenning (1991)

$$(x : \texttt{i32}) \rightarrow \{\nu : \texttt{i32} \mid \nu = x + 1\}$$

Base Type — Formula

- Types qualified with predicates that constrain the possible values of the base type
- Specifies the pre- and post-conditions of functions

Modularity: Achieved by Refinement Types

Whole-program verification

like RustHorn Matsushita et al. (2021)

```
fn prep(x: i32) -> i32
```

```
fn calc(x: i32) -> i32
```

```
fn main() {
  ...
  let v = prep(x);
  calc(v);
}
```

...

Modular verification

like Prusti Astrauskas et al. (2019)

```
#[ensures(result <= x)]
fn prep(x: i32) -> i32
```

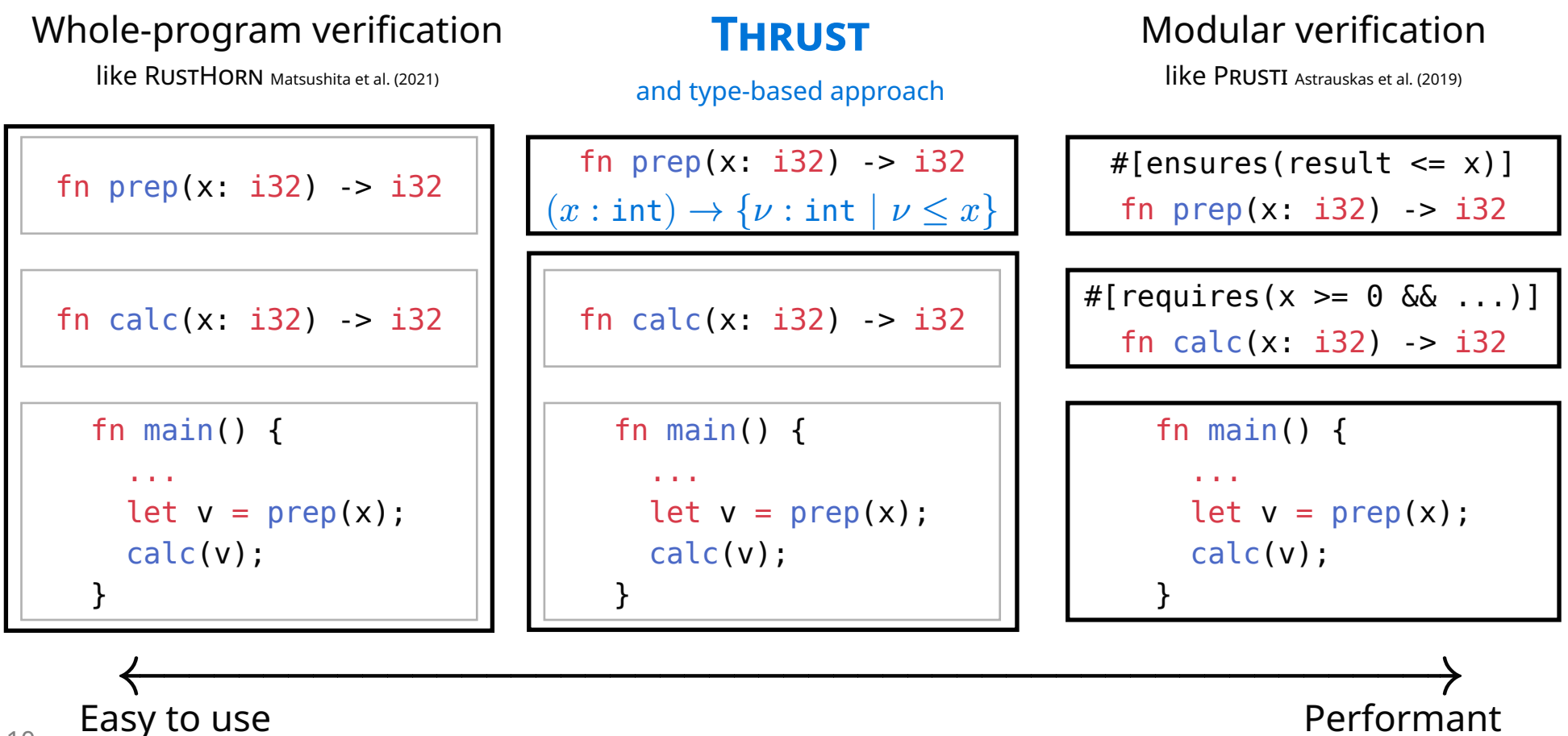
```
#[requires(x >= 0 && ...)]
fn calc(x: i32) -> i32
```

```
fn main() {
  ...
  let v = prep(x);
  calc(v);
}
```

Easy to use

Performant

Modularity: Achieved by Refinement Types



Modularity: Refinement Types

Refinement types serve as **verification boundaries**

Refinement types can also be **inferred automatically**

THRUST

and type-based approach

Modular verification

like PRUSTI Astrauskas et al. (2019)

```
fn prep(x: i32) -> i32
```

```
fn calc(x: i32) -> i32
```

```
fn main() {
  ...
  let v = prep(x);
  calc(v);
}
```

```
fn prep(x: i32) -> i32
(x : int) → {ν : int | ν ≤ x}
```

```
fn calc(x: i32) -> i32
```

```
fn main() {
  ...
  let v = prep(x);
  calc(v);
}
```

```
#[ensures(result <= x)]
fn prep(x: i32) -> i32
```

```
#[requires(x >= 0 && ...)]
fn calc(x: i32) -> i32
```

```
fn main() {
  ...
  let v = prep(x);
  calc(v);
}
```

Easy to use ⟷ Performant

Modularity: Refinement Types

Refinement types serve as **verification boundaries**

Refinement types can also be **inferred automatically**

THRUST

and type-based approach

Modular verification like PRUSTI Astrauskas et al. (2019)

```
fn prep(x: i32) -> i32
```

```
fn prep(x: i32) -> i32
```

```
#[ensures(result <= x)]
```

The type-based approach can flexibly divide the verification of a complex program into tractable parts

```
fn main() {
  ...
  let v = prep(x);
  calc(v);
}
```

```
fn main() {
  ...
  let v = prep(x);
  calc(v);
}
```

```
fn main() {
  ...
  let v = prep(x);
  calc(v);
}
```

Easy to use ⟷ Performant

Language Features: Achieved by Refinement Types

- Also, type-based abstraction naturally accommodates **higher-order functions and algebraic data types**
  - That are not supported well in other non-type-based methods

Examples:

- List of positive integers: $\mathrm{List}\langle\{\nu : \texttt{int} \mid \nu \geq 0\}\rangle$
- Higher-order specification:
  $(f : (x : \texttt{int}) \to \{\nu : \texttt{int} \mid \nu \geq x\}) \to (x : \texttt{int}) \to \{\nu : \texttt{int} \mid \nu \leq x\}$

Precision: Tracking Effects of Mutations

- Variables and memory locations in Rust programs are **mutable**
- We want to track **the effect of mutations for precision** which corresponds to what we refer to as the “precision” dimension

```
let mut x = Box::new(2);
*x = 1;
assert!(*x == 1);
```

A state change by assignment

```
let mut x = Box::new(2);
let m = &mut *x;
*m = 1;
assert!(*x == 1);
```

A state change via a mutable reference

Mutations in Refinement Type Systems

- To precisely track the values of these mutable locations, the system needs to change types in a **flow-sensitive** way

```
let mut x = Box::new(2);
// x : {x : own int | *x = 2}
*x = 1;
```

Mutations in Refinement Type Systems

- To precisely track the values of these mutable locations, the system needs to change types in a **flow-sensitive** way

```
let mut x = Box::new(2);
// x : {x : own int | *x = 2}
*x = 1;
```

Does not typecheck!

Mutations in Refinement Type Systems

- To precisely track the values of these mutable locations, the system needs to change types in a **flow-sensitive** way

```
let mut x = Box::new(2);
// x : {x : own int | *x = 2 ∨ *x = 1}
*x = 1;

assert!(*x == 1);
```

Imprecise because the assertion could not be discharged

Mutations in Refinement Type Systems

- To precisely track the values of these mutable locations, the system needs to change types in a **flow-sensitive** way

```
let mut x = Box::new(2);
// x : {x : own int | *x = 2}
*x = 1;
// x : {x : own int | *x = 1}
assert!(*x == 1);
```

Need to update the refinement

Mutations in Refinement Type Systems

- To precisely track the values of these mutable locations, the system needs to change types in a **flow-sensitive** way
- We refer to an update as a **strong update** when it changes the type in an incompatible manner

```
let mut x = Box::new(2);
// x : {x : own int | *x = 2}
*x = 1;
// x : {x : own int | *x = 1}
assert!(*x == 1);
```

Need to update the refinement (**strong update**)

Strong Updates Need Care

Allowing changes in refinements can easily lead to unsoundness

`let mut x = Box::new(2);`	$x : \{x : \texttt{own int} \mid x = \langle 2 \rangle\}$

Note: Owned pointers (`Box`) with v as referent are represented as $\langle v \rangle$

Strong Updates Need Care

Allowing changes in refinements can easily lead to unsoundness

`let mut x = Box::new(2);`	$x : \{x : \texttt{own int} \mid x = \langle 2 \rangle\}$
`let v = *x;`	$v : \{v : \texttt{int} \mid \underline{v = *x}\}$

Note: Owned pointers (`Box`) with v as referent are represented as $\langle v \rangle$

Strong Updates Need Care

Allowing changes in refinements can easily lead to unsoundness

`let mut x = Box::new(2);`	$x : \{x : \texttt{own int} \mid x = \langle 2 \rangle\}$
`let v = *x;`	$v : \{v : \texttt{int} \mid \underline{v = *x}\}$
`*x = 1;`	$x : \{x : \texttt{own int} \mid \underline{x = \langle 1 \rangle}\}$

The type of `x` changed destructively (strong update)

Note: Owned pointers (`Box`) with v as referent are represented as $\langle v \rangle$

Strong Updates Need Care

Allowing changes in refinements can easily lead to unsoundness

`let mut x = Box::new(2);`	$x : \{x : \texttt{own int} \mid x = \langle 2 \rangle\}$
`let v = *x;`	$v : \{v : \texttt{int} \mid \underline{v = *x}\}$
`*x = 1;`	$x : \left\{x : \texttt{own int} \mid \underline{x = \langle 1 \rangle}\right\}$
`assert!(v == 1);`	$\vDash \underline{v = *x = *\langle 1 \rangle} = 1$

Note: Owned pointers (`Box`) with v as referent are represented as $\langle v \rangle$

Strong Updates Need Care

Allowing changes in refinements can easily lead to unsoundness

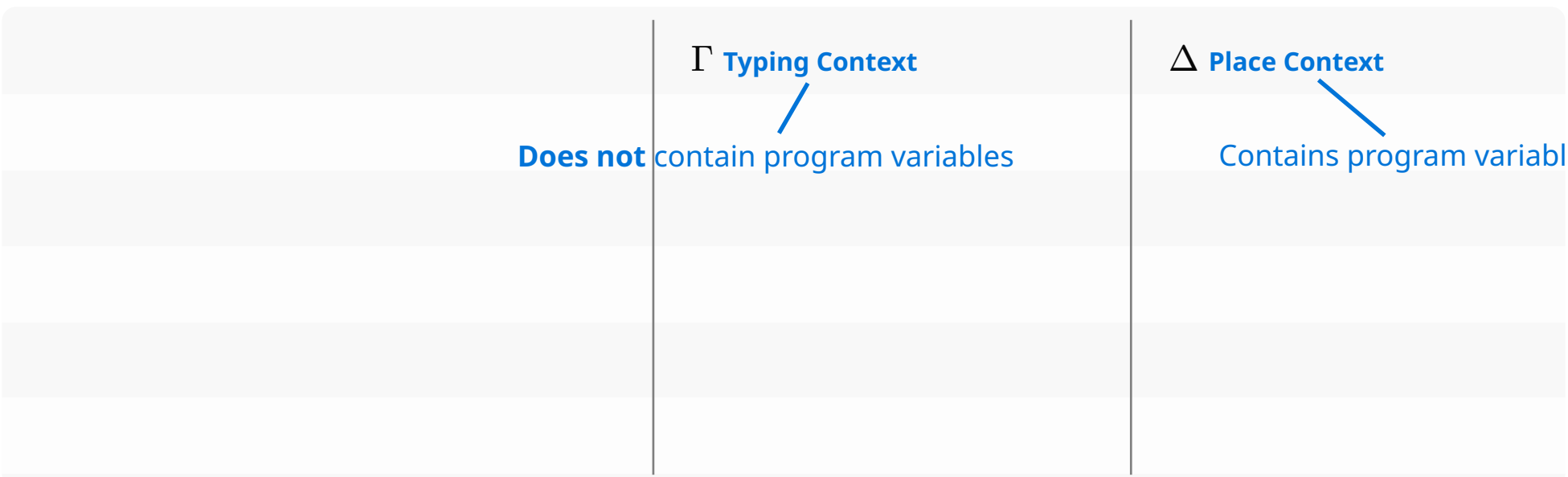
`let mut x = Box::new(2);`	$x : \{x : \texttt{own int} \mid x = \langle 2 \rangle\}$
`let v = *x;`	$v : \{v : \texttt{int} \mid \underline{v = *x}\}$
`*x = 1;`	$x : \{x : \texttt{own int} \mid \underline{x = \langle 1 \rangle}\}$
`assert!(v == 1);`	$\vDash \underline{v = *x = *\langle 1 \rangle} = 1$

However, the actual value is 2

Note: Owned pointers (`Box`) with v as referent are represented as $\langle v \rangle$

Separating the Typing Environment

- **Place Context** Δ binds program variables
- **Typing Context** Γ binds **immutable snapshot** of program variables bound in Δ, along with their refinements



Δ variables are **not allowed** to appear in Γ

Separating the Typing Environment

- **Place Context** Δ binds program variables
- **Typing Context** Γ binds **immutable snapshot** of program variables bound in Δ, along with their refinements

	Γ Typing Context	Δ Place Context
`let mut x = Box::new(2);`	$x' : \{x' : \texttt{int} \mid x' = 2\}$	$x \triangleright \langle x' \rangle$

Separating the Typing Environment

- **Place Context** Δ binds program variables
- **Typing Context** Γ binds **immutable snapshot** of program variables bound in Δ, along with their refinements

	Γ Typing Context	Δ Place Context
`let mut x = Box::new(2);`	$x' : \{x' : \texttt{int} \mid x' = 2\}$	$x \triangleright \langle x' \rangle$
`let v = *x;`		$\underline{v \triangleright x'}$

Separating the Typing Environment

- **Place Context** Δ binds program variables
- **Typing Context** Γ binds **immutable snapshot** of program variables bound in Δ, along with their re

Refinements do not change

	Γ Typing Context	Δ Place Context
`let mut x = Box::new(2);`	$x' : \{x' : \texttt{int} \mid x' = 2\}$	$x \triangleright \langle x' \rangle$
`let v = *x;`		$v \triangleright x'$
`*x = 1;`	$x'' : \{x'' : \texttt{int} \mid x'' = 1\}$	$x \triangleright \langle x'' \rangle$

Value of x swapped

Separating the Typing Environment

- **Place Context** Δ binds program variables
- **Typing Context** Γ binds **immutable snapshot** of program variables bound in Δ, along with their refinements

	Γ Typing Context	Δ Place Context
`let mut x = Box::new(2);`	$x' : \{x' : \texttt{int} \mid x' = 2\}$	$x \triangleright \langle x' \rangle$
`let v = *x;`		$\underline{v \triangleright x'}$
`*x = 1;`	$x'' : \{x'' : \texttt{int} \mid x'' = 1\}$	$x \triangleright \langle x'' \rangle$
`assert!(*x == 1);`	$\vDash *x = x'' = 1$	
`// assert!(v == 1);`	$\nvDash \underline{v = x'} = 1$	

Separating the Typing Environment

- **Place Context** Δ binds program variables
- **Typing Context** Γ binds **immutable snapshot** of program variables bound in Δ, along with their refinements

We addressed the unsoundness issue by
separating the flow-sensitive part of the environment

let		
let v = *x;		$\underline{v \triangleright x}$
*x = 1;	$x'' : \{x'' : \texttt{int} \mid x'' = 1\}$	$x \triangleright \langle x'' \rangle$
assert!(*x == 1);	$\vDash *x = x'' = 1$	
// assert!(v == 1);	$\nvDash \underline{v = x'} = 1$	

Challenge in Supporting Mutable References

- **Mutable references** are essential for practical Rust programming
- To support mutable references in our type system, we have to **propagate changes** made through the references to their owners

```
let mut x = Box::new(2);
let m = &mut *x;
// m: { m: &mut int | *m = 2 }
decr(m);
// m: { m: &mut int | *m = 1 }
assert!(*m == 1);
assert!(*x == 1);
```

```
fn decr(m: &mut i32) {
  *m -= 1;
}
```

Challenge in Supporting Mutable References

- **Mutable references** are essential for practical Rust programming
- To support mutable references in our type system, we have to **propagate changes** made through the references to their owners

```
let mut x = Box::new(2);
let m = &mut *x;
// m: { m: &mut int | *m = 2 }
decr(m);
// m: { m: &mut int | *m = 1 }
assert!(*m == 1);
assert!(*x == 1);
```

```
fn decr(m: &mut i32) {
  *m -= 1;
}
```

The assertion fails if x left unchanged

The propagation is not achieved by a simple strong update

Prophecies to Track Mutable References

- The use of an auxiliary variable called a **prophecy variable** Abadi and Lamport (1991) enables reasoning about future states
- RustHorn Matsushita et al. (2021) leveraged prophecy variables to implement this propagation and we incorporated a similar idea to handle mutable references
  - Still **we needed to adapt it because RustHorn is not a type-based formalization**

Prophecies in Refinement Types

```
fn decr(m: &mut i32) {
  *m -= 1;
}
```

$$(m : \&\texttt{mut i32}) \to \{\text{unit} \mid \circ m = *m - 1\}$$

- Mutable references are represented by a pair of values: $\langle a, l \rangle$
  - When $m = \langle a, l \rangle$, $*m = a$ and $\circ m = l$
  - a is current value of the referent
  - l is final value of the referent, at the end of the reference
    - We refer to this as the **prophecy** of `m`
- Just a simple tuple, so **well-suited for automation**

Prophecies in Refinement Types

```
fn decr(m: &mut i32) {
  *m -= 1;
}
```

`m` at the entry of the call

`m` at the end of the call

$$(m : \&\texttt{mut i32}) \to \{\text{unit} \mid \circ m = *m - 1\}$$

- Mutable references are represented by a pair of values: $\langle a, l \rangle$
  - When $m = \langle a, l \rangle$, $*m = a$ and $\circ m = l$
  - a is current value of the referent
  - l is final value of the referent, at the end of the reference
    - We refer to this as the **prophecy** of `m`
- Just a simple tuple, so **well-suited for automation**

Prophecies Link Owners and References

	Γ Typing Context	Δ Place Context
`let mut x = Box::new(2);`	$x' : \{\nu : \texttt{int} \mid \nu = 2\}$	$x \triangleright \langle x' \rangle$

where $\mathrm{decr} : (a : \&\texttt{mut int}) \to \{\mathrm{unit} \mid \circ a = *a - 1\}$

Prophecies Link Owners and References

	Γ Typing Context	Δ Place Context
`let mut x = Box::new(2);`	$x' : \{\nu : \texttt{int} \mid \nu = 2\}$	$x \triangleright \langle x' \rangle$
`let m = &mut *x;`	$l : \texttt{int}$	$x \triangleright \langle l \rangle, m \triangleright \langle x', l \rangle$

A prophecy variable, freshly defined upon `&mut` in a flow-sensitive manner

Value of `x` swapped

where $\mathrm{decr} : (a : \&\texttt{mut int}) \to \{\mathrm{unit} \mid \circ a = *a - 1\}$

Prophecies Link Owners and References

	Γ Typing Context	Δ Place Context
`let mut x = Box::new(2);`	$x' : \{\nu : \texttt{int} \mid \nu = 2\}$	$x \triangleright \langle x' \rangle$
`let m = &mut *x;`	$l : \texttt{int}$	$x \triangleright \langle l \rangle, m \triangleright \langle x', l \rangle$
`decr(m);`	$\_ : \{ \circ\langle x', l\rangle = *\langle x', l\rangle - 1 \}$	

Substituted canonical form of `m` in Δ for a

where $\text{decr} : (a : \&\texttt{mut int}) \to \{\text{unit} \mid \circ a = *a - 1\}$

Prophecies Link Owners and References

	Γ Typing Context	Δ Place Context
`let mut x = Box::new(2);`	$x' : \{\nu : \texttt{int} \mid \nu = 2\}$	$x \triangleright \langle x' \rangle$
`let m = &mut *x;`	$l : \texttt{int}$	$x \triangleright \langle l \rangle, m \triangleright \langle x', l \rangle$
`decr(m);`	$\_ : \{\, l = x' - 1 \,\}$	
`assert!(*x == 1);`	$\vDash *\langle l \rangle = l = x' - 1 = 1$	

$x \triangleright \langle l \rangle$ in Δ

where $\mathrm{decr} : (a : \&\texttt{mut int}) \to \{\mathrm{unit} \mid \circ a = *a - 1\}$

Soundness under “Aliasing XOR Mutability”

- We prove the soundness of our type system under the assumption that any execution of **the program is “well-borrowed”**
  - Well-borrowedness is a formalization of “aliasing XOR mutability” based on the Stacked Borrows Jung et al. (2020) aliasing model (we adapted for our formalization)
  - The borrow checker statically guarantees this for safe Rust
- Thus, our soundness does not depend on the specific behavior or implementation of the borrow checker / the underlying type system
  - This **allows for the evolution of the type checker in Rust** such as two-phase borrows, non-lexical lifetimes (NLL), and Polonius

Automation:
Achieved by CHC Solving Unno and Kobayashi (2009)

Our primary concern is the inference of **inductive invariants**

```
fn repeat(n: i32, ma: &mut i32, mb: &mut i32) {
  if n != 0 {
    let mc = take(ma, mb);
    *mc += 1;
    repeat(n - 1, ma, mb);
  }
}
fn main() {
  let n = rand();
  let mut a = rand();
  let mut b = rand();
  repeat(n, &mut a, &mut b);
  assert!(a - b >= n || b - a >= n);
}
```

Automation:
Achieved by CHC Solving Unno and Kobayashi (2009)

Our primary concern is the inference of **inductive invariants**

```
fn repeat(n: i32, ma: &mut i32, mb: &mut i32) {
  if n != 0 {
    let mc = take(ma, mb);
    *mc += 1;
    repeat(n - 1, ma, mb);
  }
}
fn main() {
  let n = rand();
  let mut a = rand();
  let mut b = rand();
  repeat(n, &mut a, &mut b);
  assert!(a - b >= n || b - a >= n);
}
```

1. Generate template

$$(\mathrm{n} : \texttt{i32}, \mathrm{ma} : \texttt{\&mut i32}, \mathrm{mb} : \texttt{\&mut i32}) \to \{\mathrm{unit} \mid p_1(\mathrm{n}, \mathrm{ma}, \mathrm{mb})\}$$

Automation:
Achieved by CHC Solving Unno and Kobayashi (2009)

Our primary concern is the inference of **inductive invariants**

```
fn repeat(n: i32, ma: &mut i32, mb: &mut i32) {
  if n != 0 {
    let mc = take(ma, mb);
    *mc += 1;
    repeat(n - 1, ma, mb);
  }
}
fn main() {
  let n = rand();
  let mut a = rand();
  let mut b = rand();
  repeat(n, &mut a, &mut b);
  assert!(a - b >= n || b - a >= n);
}
```

1. Generate template

$$(\mathrm{n} : \texttt{i32}, \mathrm{ma} : \texttt{\&mut i32}, \mathrm{mb} : \texttt{\&mut i32}) \to \{\mathrm{unit} \mid p_1(\mathrm{n}, \mathrm{ma}, \mathrm{mb})\}$$

2. Generate constraints

$$\begin{aligned}
p_1(v_0, v_1, v_2) &\Leftarrow p_2(v_0, v_1, v_2) \\
p_2(v_0, v_1, v_2) &\Leftarrow p_3(v_3, v_4, v_5, v_6, v_7) \wedge v_0 = v_3 \\
&\quad \wedge v_1 = \langle v_5, v_4 \rangle \wedge v_2 = \langle v_7, v_6 \rangle \\
p_3(v_0, v_1, v_2, v_3, v_4) &\Leftarrow p_4(\langle v_4, v_3 \rangle, v_0, \langle v_2, v_1 \rangle) \wedge v_0 \neq 0 \\
&\quad \wedge v_5 = v_0 \wedge v_6 = \langle v_2, v_1 \rangle \\
&\quad \wedge v_7 = \langle v_4, v_3 \rangle \wedge p_5(v_5, v_6, v_7) \\
&\quad \vdots
\end{aligned}$$

Automation:
Achieved by CHC Solving Unno and Kobayashi (2009)

Our primary concern is the inference of **inductive invariants**

```
fn repeat(n: i32, ma: &mut i32, mb: &mut i32) {
  if n != 0 {
    let mc = take(ma, mb);
    *mc += 1;
    repeat(n - 1, ma, mb);
  }
}
fn main() {
  let n = rand();
  let mut a = rand();
  let mut b = rand();
  repeat(n, &mut a, &mut b);
  assert!(a - b >= n || b - a >= n);
}
```

1. Generate template

$$(\mathrm{n} : \texttt{i32}, \mathrm{ma} : \texttt{\&mut i32}, \mathrm{mb} : \texttt{\&mut i32}) \to \{\mathrm{unit} \mid p_1(\mathrm{n}, \mathrm{ma}, \mathrm{mb})\}$$

2. Generate constraints

$$\begin{aligned}
p_1(v_0, v_1, v_2) &\Leftarrow p_2(v_0, v_1, v_2) \\
p_2(v_0, v_1, v_2) &\Leftarrow p_3(v_3, v_4, v_5, v_6, v_7) \wedge v_0 = v_3 \\
&\quad \wedge v_1 = \langle v_5, v_4 \rangle \wedge v_2 = \langle v_7, v_6 \rangle \\
p_3(v_0, v_1, v_2, v_3, v_4) &\Leftarrow p_4(\langle v_4, v_3 \rangle, v_0, \langle v_2, v_1 \rangle) \wedge v_0 \neq 0 \\
&\quad \wedge v_5 = v_0 \wedge v_6 = \langle v_2, v_1 \rangle \\
&\quad \wedge v_7 = \langle v_4, v_3 \rangle \wedge p_5(v_5, v_6, v_7) \\
&\quad \vdots
\end{aligned}$$

3. Solve

$$\begin{aligned}
p_1(\mathrm{n}, \mathrm{ma}, \mathrm{mb}) :\Leftrightarrow (&n = 0 \\
&\wedge *\mathrm{ma} = \circ\mathrm{ma} \\
&\wedge *\mathrm{mb} = \circ\mathrm{mb}) \\
&\vee ((*\mathrm{ma} < *\mathrm{mb} \\
&\vee (n - \circ\mathrm{ma} + \circ\mathrm{mb} + *\mathrm{ma} - *\mathrm{mb}) < 1) \\
&\wedge \ldots)
\end{aligned}$$

Automation:
Achieved by CHC Solving Unno and Kobayashi (2009)

Our primary concern is the inference of **inductive invariants**

```
fn repeat(n: i32, ma: &mut i32, mb: &mut i32) {
  if n
    let
    *mc
    rep
  }
}
fn main
  let n
  let mut a = rand();
  let mut b = rand();
  repeat(n, &mut a, &mut b);
  assert!(a - b >= n || b - a >= n);
}
```

1 Generate template

THRUST could readily utilize CHC solving to automate reasoning about pointers, thanks to its prophecy-based formalization

$$
\begin{aligned}
p_1(v_0, v_1, v_2) &\Leftarrow p_2(v_0, v_1, v_2) \\
p_2(v_0, v_1, v_2) &\Leftarrow p_3(v_3, v_4, v_5, v_6, v_7) \wedge v_0 = v_3 \\
&\quad \wedge v_1 = \langle v_5, v_4 \rangle \wedge v_2 = \langle v_7, v_6 \rangle \\
p_3(v_0, v_1, v_2, v_3, v_4) &\Leftarrow p_4(\langle v_4, v_3 \rangle, v_0, \langle v_2, v_1 \rangle) \wedge v_0 \neq 0 \\
&\quad \wedge v_5 = v_0 \wedge v_6 = \langle v_2, v_1 \rangle \\
&\quad \wedge v_7 = \langle v_4, v_3 \rangle \wedge p_5(v_5, v_6, v_7) \\
&\quad \vdots
\end{aligned}
$$

$$
\begin{aligned}
p_1(\mathrm{n}, \mathrm{ma}, \mathrm{mb}) :\Leftrightarrow\ & (n = 0 \\
& \wedge *\mathrm{ma} = \circ\mathrm{ma} \\
& \wedge *\mathrm{mb} = \circ\mathrm{mb}) \\
& \vee ((*\mathrm{ma} < *\mathrm{mb} \\
& \vee (n - \circ\mathrm{ma} + \circ\mathrm{mb} + *\mathrm{ma} - *\mathrm{mb}) < 1) \\
& \wedge \ldots)
\end{aligned}
$$

Evaluation

We conducted two experiments using two different benchmark sets to confirm whether Thrust's design goals are achieved

- RQ. 1: Has Thrust actually achieved its design goal?
- RQ. 2: How does the formalization difference from RustHorn affect the verification performance?
  - RustHorn is another CHC-based verifier; it differs from Thrust in that it is formalized as a translation from Rust into CHC

New Benchmark Set for RQ. 1

- We created **a new benchmark set** designed to compare a broader range of verifiers including those that are not limited to automated verification tools
  - Four categories: Modularity, Higher-Order, ADTs, and Precision
  - We have 8 (including safe / unsafe variants) cases for each category = a total of 32 cases
- We evaluated Thrust against other notable Rust verification tools: RustHorn, Flux, and Prusti

Experimental Results for RQ. 1 (Excerpt)

(Showing 4 out of 32 results, selecting particularly distinctive ones)

	Thrust			RustHorn Matsushita et al. (2021)		Flux Lehmann et al. (2023)			Prusti Astrauskas et al. (2019)		
	Annot.	Result	Time(s)	Result	Time(s)	Annot.	Result	Time(s)	Annot.	Result	Time(s)
Modularity nonlin-sum-safe	2	✓	0.05	?‡	N/A	1	✓	0.09	2	✓	4.09
Higher-Order list-fold-safe	0	✓	0.11	?†	N/A	0	?	0.06	0	?†	N/A
ADTs list-pos-safe	0	✓	0.28	?‡	N/A	2	✓	0.12	5	✓	6.69
Precision inc-max-plain-safe	0	✓	0.05	✓	0.04	0	?	0.07	0	✓	3.77

? The tool failed to prove safety

† The tool was aborted

‡ The tool timed out

Experimental Results for RQ. 1 (Excerpt)

(Showing 4 out of 32 results, selecting particularly distinctive ones)

	Thrust			RustHorn (Matsushita et al. (2021))		Flux (Lehmann et al. (2023))			Prusti (Astrauskas et al. (2019))		
	Annot.	Result	Time(s)	Result	Time(s)	Annot.	Result	Time(s)	Annot.	Result	Time(s)
Modularity nonlin-sum-safe	2	✓	0.05	?‡	N/A	1	✓	0.09	2	✓	4.09
Higher-Order list-fold-safe	0	✓	0.11	?†	N/A	0	?	0.06	0	?†	N/A
ADTs list-pos-safe	0	✓	0.28	?‡	N/A	2	✓	0.12	5	✓	6.69
Precision inc-max-plain-safe	0	✓	0.05	✓	0.04	0	?	0.07	0	✓	3.77

? The tool failed to prove safety
† The tool was aborted
‡ The tool timed out

- Thrust, Flux, and Prusti all **support modular verification**
- Only Thrust can **handle higher-order functions**
- Thrust **requires fewer annotations** in ADT-related programs
-

Experimental Results for RQ. 1 (Excerpt)

(Showing 4 out of 32 results, selecting particularly distinctive ones)

	Thrust			RustHorn Matsushita et al. (2021)		Flux Lehmann et al. (2023)			Prusti Astrauskas et al. (2019)		
	Annot.	Result	Time(s)	Result	Time(s)	Annot.	Result	Time(s)	Annot.	Result	Time(s)
Modularity nonlin-sum-safe	2	✓	0.05	?‡	N/A	1	✓	0.09	2	✓	4.09
Higher-Order list-fold-safe	0	✓	0.11	?†	N/A	0	?	0.06	0	?†	N/A
ADTs list-pos-safe	0	✓	0.28	?‡	N/A	2	✓	0.12	5	✓	6.69
Precision inc-max-plain-safe	0	✓	0.05	✓	0.04	0	?	0.07	0	✓	3.77

? The tool failed to prove safety
† The tool was aborted
‡ The tool timed out

- Thrust, Flux, and Prusti all **support modular verification**
- Only Thrust can **handle higher-order functions**
- Thrust **requires fewer annotations** in ADT-related programs
- Thrust is **more precise** than Flux

Experimental Results for RQ. 1 (Excerpt)

(Showing 4 out of 32 results, selecting particularly distinctive ones)

	Thrust			RustHorn (Matsushita et al. (2021))		Flux (Lehmann et al. (2023))			Prusti (Astrauskas et al. (2019))		
	Annot.	Result	Time(s)	Result	Time(s)	Annot.	Result	Time(s)	Annot.	Result	Time(s)
Modularity nonlin-sum-safe	2	✓	0.05	?‡	N/A	1	✓	0.09	2	✓	4.09
Higher-Order list-fold-safe	0	✓	0.11	?†	N/A	0	?	0.06	0	?†	N/A
ADTs list-pos-safe	0	✓	0.28	?‡	N/A	2	✓	0.12	5	✓	6.69
Precision inc-max-plain-safe	0	✓	0.05	✓	0.04	0	?	0.07	0	✓	3.77

? The tool failed to prove sa
† The tool was aborted
‡ The tool timed out

Thrust has achieved its design goal
(whereas other tools do not meet our design goals)

Benchmark Case: Precision

```
fn main() {
  let mut a = rand();
  let mut b = rand();
  {
    let ma = &mut a;
    let mb = &mut b;
    let mc = if *ma >= *mb { ma } else { mb };
    *mc += 1;
  }
  assert!(a != b);
}
```

THRUST provides higher precision than FLUX for programs that involve dynamically selected mutable references

Benchmark Case: Polymorphic ADTs

```rust
fn make_list(n: i32) -> List<i32> {
  if rand_bool() {
    List::Cons(n, Box::new(make_list(n)))
  } else {
    List::Nil
  }
}
fn pick(la: &List<i32>) -> i32 {
  match la {
    List::Cons(x, lb) =>
      if rand_bool() { *x }
      else { pick(lb) },
    List::Nil => loop {},
  }
}
```

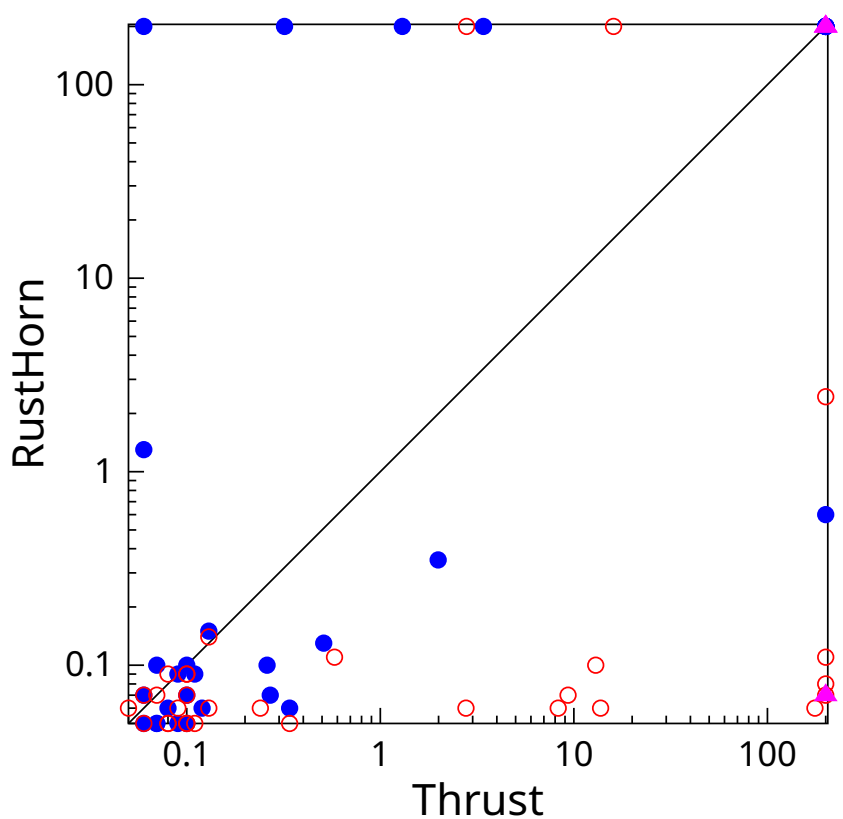
```rust
fn main() {
    let n = rand();
    let n = if n < 0 { -n } else { n };
    let list = make_list(n);
    assert!(pick(&list) >= 0);
}
```

THRUST handles polymorphic ADTs more effectively, offering:

- **Better performance than RUSTHORN**
- **Less annotation than PRUSTI**

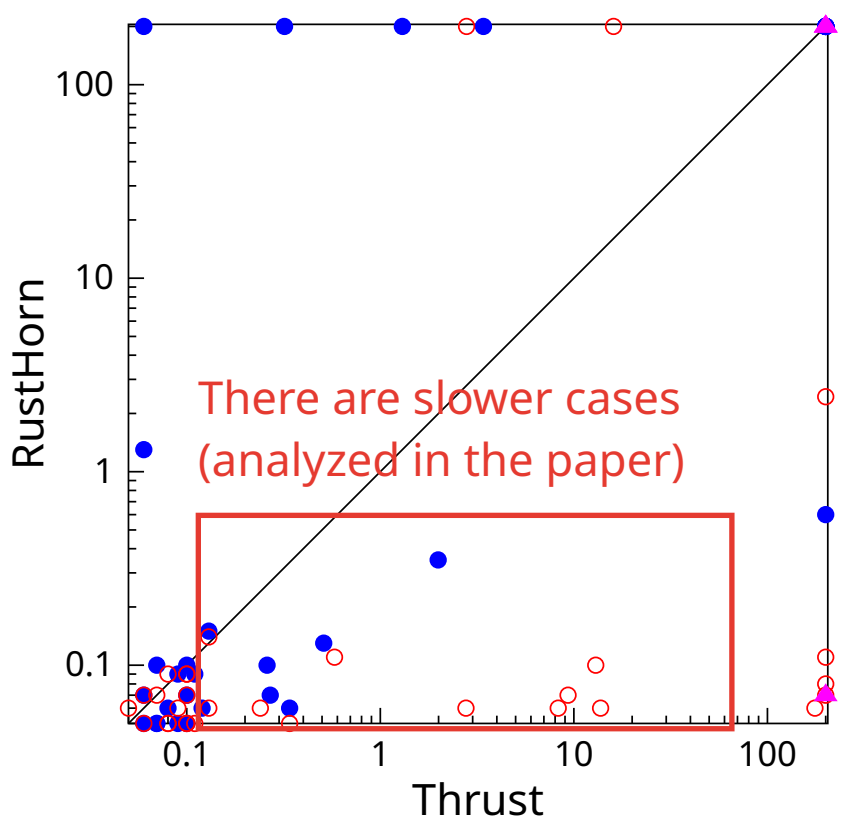
Experimental Results for RQ. 2

- Comparison with RustHorn using RustHorn's benchmark set (a total of 73 cases)
- In most cases, **verification with Thrust was completed in a comparable time**
  - There are cases that time out only with RustHorn, as well as cases that time out only with Thrust



Experimental Results for RQ. 2

- Comparison with RustHorn using RustHorn's benchmark set (a total of 73 cases)
- In most cases, **verification with Thrust was completed in a comparable time**
  - There are cases that time out only with RustHorn, as well as cases that time out only with Thrust



Conclusion

We presented THRUST, an **automated**, **modular**, and **precise** verification tool for safe RUST

1. We integrated (technical challenge):
   Refinement Types + **CHC Solving** + **Flow-Sensitivity** + **Prophecies**
2. We designed a novel refinement type system and proved its soundness under the assumption of “aliasing XOR mutability”
3. We implemented and evaluated the system against other notable RUST verification tools, and obtained promising results

Implementation is available at https://github.com/coord-e/thrust