Thrust: A Prophecy-based Refinement Type System for Rust

HIROMI OGAWA, University of Tsukuba, Japan
TARO SEKIYAMA, National Institute of Informatics, Japan and SOKENDAI, Japan
HIROSHI UNNO, Tohoku University, Japan

We introduce Thrust, a new verification tool for ensuring functional correctness in Rust, distinguished by its strengths in *automated* verification, including the synthesis of inductive invariants for loops and recursive functions. Thrust is built on a novel dependent refinement type system for Rust and refinement type inference techniques based on Constrained Horn Clause (CHC) solvers. Leveraging advantages of the type system, Thrust also supports semi-automated verification utilizing user type annotations to complement CHC solvers in cases where automatic constraint solving is unsuccessful, as well as *modular* verification at the function and subexpression levels. Thrust also achieves *precise* verification, especially for programs involving pointer aliasing and borrowing, without sacrificing the benefits of automated verification, by incorporating the notion of *prophecy* into the refinement type system: it not only enables strong updates by leveraging the "aliasing XOR mutability" guarantee provided by Rust's type system, but also achieves propagation of update information to the original owner upon mutable borrow release through the use of a prophecy variable. Incorporating prophecy into a refinement type system is itself challenging and requires certain tricks, as discussed in this paper, making a theoretical contribution and paving the way for further research into prophecy-based refinement type systems. While our type system addresses the challenge, we keep it simple for extensibility, specifically by delegating the guarantee of "aliasing XOR mutability," and, more technically, the "well-borrowedness" of the program in the sense of the *stacked borrows* aliasing model, to Rust's type system, allowing us to focus on reasoning about functional correctness and propagating update information through prophecy variables. Compared to RustHorn, another automated verification tool based on prophecy, our approach leverages the strengths of refinement types to support modular verification, higher-order functions, and refinement of data stored in algebraic data structures. We implemented Thrust, a refinement type inference tool as a plugin for the Rust compiler, and evaluated it using RustHorn benchmarks, as well as additional new benchmarks, including those that are beyond the capabilities of RustHorn and other semi-automated verification tools, obtaining promising results.

CCS Concepts: • **Theory of computation → Logic and verification**; **Type theory**; • **Software and its engineering → Formal software verification**; **Functional languages**.

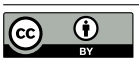
Additional Key Words and Phrases: Rust, borrowing, prophecy, refinement types, constrained Horn clauses

ACM Reference Format:
Hiromi Ogawa, Taro Sekiyama, and Hiroshi Unno. 2025. Thrust: A Prophecy-based Refinement Type System for Rust. *Proc. ACM Program. Lang.* 9, PLDI, Article 230 (June 2025), 29 pages. https://doi.org/10.1145/3729333

1 Introduction

Rust [15] is a general-purpose programming language widely adopted across industries, ranging from the Linux kernel [61] to the world's largest software registry [18]. Like C and C++, Rust allows low-level memory operations, making it well-suited for system programming and enabling high-performance applications. At the same time, it ensures memory safety through its unique

Authors' Contact Information: Hiromi Ogawa, University of Tsukuba, Tsukuba, Japan, coord_e@coins.tsukuba.ac.jp; Taro Sekiyama, National Institute of Informatics, Tokyo, Japan and SOKENDAI, Tokyo, Japan, tsekiyama@acm.org; Hiroshi Unno, Tohoku University, Sendai, Japan, hiroshi.unno@acm.org.



ACM 2475-1421/2025/6-ART230
https://doi.org/10.1145/3729333

Table 1. Comparison of Static Verification Tools for "safe" Rust. The symbols †, ‡, and ¶ indicate partial "yes" answers, and further details are discussed in the main text.

	Automated?	Modular?	Precise?	Datatypes?	Higher-Order Functions?
Thrust	✓	✓	✓	✓	✓
RustHorn [47]	✓		✓	¶	
Creusot [17]		✓	✓	✓	✓
Flux [43]	†	✓	‡	✓	
RefinedRust [25]		✓	✓	✓	
Prusti [3]		✓	‡	✓	
Electrolysis [63]		✓		✓	✓
Aeneas [29, 30]		✓		✓	
Verus [42]		✓		✓	✓

type system based on ownership, borrowing, and lifetimes, addressing the issues of unsafe memory handling that are common in languages like C and C++. Rust also emphasizes developer productivity through its language features and tooling, making it popular not only among systems programmers but also across a wide range of software development fields.

Driven by the growing popularity of Rust, various static verification tools have been proposed, including Electrolysis [63], Prusti [3], RustHorn [47], Creusot [17], Aeneas [29, 30], Flux [43], Verus [42], and RefinedRust [25], which leverage the "aliasing XOR mutability" guarantee provided by Rust's type system. This guarantee ensures that a given memory location can either have multiple references (aliasing) without modification (immutability), or a single mutable reference, but not both simultaneously, which helps in verifying higher-level properties such as functional correctness of "safe" Rust programs, meaning programs that do not use the `unsafe` keyword. These tools have successfully been applied to various real-world Rust programs. Tools like RefinedRust, RustBelt [33], and RustHornBelt [46] extend their capabilities to allow verification of Rust code marked as `unsafe` using a sophisticated memory-aware underlying logic based on Iris [34]. On the other hand, this paper focuses on "safe" Rust programs that follow the aliasing discipline to aid higher-level verification.

These tools, however, each have specific pros and cons in terms of *automation*, *modularity*, *precision*, and *supported language features*, as summarized in Table 1. Despite their individual strengths, none provide a balanced solution that fully addresses all these critical aspects. To fill this gap, this paper proposes Thrust, a new verification tool based on refinement types [24, 73] and designed to combine strengths across all these dimensions, aiming to achieve a more comprehensive and effective solution for verifying Rust programs. In the following, we compare Thrust with existing tools regarding the aspects mentioned above, clarifying the design goals of Thrust, which will be further discussed in detail with concrete examples in Section 2.

1.1 Design Goals of Thrust

Automated verification. Automated synthesis of inductive invariants is essential to reduce the annotation burden on users when verifying loops and recursive functions. Techniques for refinement type inference based on Constrained Horn Clause (CHC) solving (see, e.g., [9]), as studied by Unno and Kobayashi [64] and Rondon et al. [52] for ML-like functional programs, have laid the foundation. Building on these techniques in the context of Rust verification, Thrust achieves automation through CHC-based refinement type inference to synthesize inductive invariants in Rust programs. By contrast, RustHorn achieves a similar level of automation by directly reducing verification to

CHC solving without relying on refinement types. Meanwhile, Flux employs a specialized form of CHC solving called liquid type inference, based on the Houdini algorithm [23], which achieves a relatively high degree of automation. However, as indicated by † in the table, Flux requires a predefined set of predicates as input for synthesis. Additionally, to enable strong updates, users of Flux must use the custom extension *strong reference* `&strg` instead of the mutable reference `&mut`. In Flux's type system, strong references are tracked as singleton types. However, since automatic type inference for functions that take a strong reference is not possible, their summaries need to be annotated as refinement types. While Prusti reduces verification problems to the Viper verification framework [48] based on separation logic and further reduces to SMT solving, Verus employs lightweight linear type checking to assist in reducing to SMT solving, and RefinedRust reduces to Coq (especially using Iris [34]). Meanwhile, Electrolysis, Aeneas, and Creusot convert Rust programs into pure functional programs expressed as terms in Lean, Coq, and WhyML, the programming language of the Why3 verification framework [22], respectively. These approaches require user annotations of inductive invariants for loops and recursive functions to generate verification conditions and/or mechanized but manual proofs to discharge them.

Modular verification. Because fully automated synthesis of inductive invariants is not guaranteed to always succeed, modular verification, which involves breaking down verification by program components to reduce complexity, is essential. In this context, user annotations serve as hints to the backend solver, and they are necessary to scale verification to large programs. Accordingly, Thrust enables modular verification through its refinement type system, supporting function- and subexpression-level modular verification and allowing user annotations as hints to its backend CHC solver. By contrast, RustHorn lacks a program logic for the source program, which limits modular verification and prevents the use of user annotations as hints. Prusti, Flux, and RefinedRust achieve modular verification through their program logics. Additionally, because Electrolysis, Aeneas, Creusot, and Verus also translate specifications and annotations from the source program, they can be regarded as modular verification condition generators.

Precise verification. In languages like Rust, which allow low-level memory manipulation through pointers and references, precise reasoning about runtime values becomes challenging for static verification tools when only weak updates are possible due to pointer aliasing. Rust's ownership type system provides the "aliasing XOR mutability" guarantee, making it possible and essential for static verification tools to permit strong updates to achieve precise verification in functional correctness checks. Rust also introduces features like borrowing and lifetimes to add flexibility to the ownership system. However, for precise verification, it becomes crucial to propagate update information to the original owner when a mutable borrow is released. The concept of *prophecy* [1] has been effectively applied in Rust verification tools such as RustHorn, Creusot, and RefinedRust (where prophecy variables are referred to as *borrow names*) to facilitate this information propagation. Inspired by RustHorn, Thrust also leverages prophecy to achieve precise verification, especially for programs with pointer aliasing and borrowing, without sacrificing full automation. This is made possible by a novel dependent refinement type system that enables strong updates through Rust's "aliasing XOR mutability" guarantee. Using prophecy variables, Thrust propagates update information to the original owner upon mutable borrow release, leveraging the "well-borrowedness" guarantee, in the sense of the *stacked borrows* aliasing model [32], provided by Rust's type system. Although tools like Prusti and Flux achieve a high level of precision, as indicated by ‡ in the table, their lack of prophecy leads to challenges in verifying programs that require propagating update information to the original owner when a mutable borrow is released. Flux loses precision where ownership changes depending on runtime values, particularly in cases where variables that are dropped, and thus the ownership, differ among the branches of an if expression or a pattern match

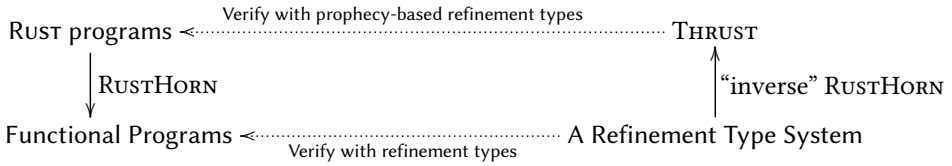
expression. Prusti has a mechanism called *pledges* to describe the behavior of references returned from functions. However, at least in the current implementation of Prusti, this mechanism is less expressive than prophecy variables; for instance, it cannot accommodate the splitting of borrows.

Supported language features. The Rust language supports various features originating from functional languages, such as higher-order functions, algebraic data types, type classes (or traits), and modules. Because Thrust is designed based on a refinement type system for an ML-like functional language, it can naturally handle higher-order functions and algebraic data types. By contrast, RustHorn does not support higher-order functions and lacks the ability to express invariants of data elements stored in algebraic data structures like lists and trees, as indicated by ¶ in the table. Thrust, on the other hand, allows compositional verification by inferring refinement types for higher-order functions and data elements within algebraic data types. In particular, the latter reduces the burden on the backend CHC solver, which would otherwise need to synthesize recursive functions over algebraic data types to represent necessary invariants for verification, often resulting in challenging CHC constraints solvable only by a certain CHC solver. By supporting refinement types for data elements, Thrust can employ a variety of CHC solvers that support algebraic data types, allowing for more flexible verification.

1.2 Prophecy-based Refinement Type System of Thrust

We now briefly introduce the design philosophy behind Thrust's prophecy-based refinement type system. Over the years, refinement type systems for ML-like functional languages [6, 24, 52, 64, 73] have been extended. These extensions have focused on language features, particularly expanding support for mutable cells [4, 25, 27, 43, 49, 53, 54, 60, 74] and other effectful computations [37, 41, 55, 57, 69]. Additional developments have included improving analysis precision [38, 66], as well as extending support for advanced specifications, such as termination and temporal liveness properties [31, 50, 72], and for falsifying (in addition to verifying) safety and termination properties [65].

In designing the refinement type system for Thrust, instead of basing it on an existing refinement type system that supports mutable cells, as Flux did with [53], we took a different approach. We designed our refinement type system for Rust by considering the "inverse" of the translation from Rust programs to pure functional programs, as realized in RustHorn and Creusot. By pulling back a refinement type system for a pure functional language to Rust, we designed Thrust's prophecy-based refinement type system tailored for Rust.[1]



Our approach has the following advantages.

- The first is the precision of the resulting refinement type system, which is achieved by building on the sound and complete translation of RustHorn, which itself uses the concept

[1] RefinedRust [25], which was concurrently developed with this research, can also be considered a prophecy-based refinement type system. However, while Thrust emphasizes automated verification based on CHC solving, RefinedRust focuses on foundational proofs based on Iris logic, marking a significant difference in design philosophy. Additionally, the RefinedRust paper does not include typing rules, and the technical intricacies of incorporating prophecy, which are discussed in this paper, are not addressed.

of prophecy to precisely propagate the update information of mutable borrows to the original owner. Like RustHorn, Thrust represents a mutable borrow as a pair of values: the current value of the borrow and the prophecy, which plays a key role in propagating the value, when the mutable borrow is released, to the original owner. From the time the borrow is created until it is released, the prophecy refers to the "future value" of the mutable borrow, which is the value when the borrow is released and is not yet known. This simple representation enables precise and automated verification without the need for more detailed representations of heap states, such as those used in separation logic.

- The second advantage is its high extensibility. Given the various extensions mentioned earlier for the original refinement type systems, we believe our approach enables supporting a wide range of language features and advanced specifications. In this paper, we primarily leveraged this advantage to support language features such as higher-order functions and algebraic data types.

Incorporating the concept of prophecy into a refinement type system is itself challenging and requires certain tricks, as discussed in Section 4, making a theoretical contribution and paving the way for further research into prophecy-based refinement type systems. While our type system addresses the challenge, we keep it simple for extensibility, specifically by delegating the guarantee of "aliasing XOR mutability," and, more technically, the "well-borrowedness" of the program, to Rust's type system, allowing us to focus on reasoning about functional correctness and propagating update information through prophecies.

The contributions of this paper are:

- We formalize a core language for Rust and the prophecy-based refinement type system that supports (im)mutable borrows, nested and partial (re)borrows, higher-order functions, and algebraic data types, providing a theoretical foundation for Thrust.
- We prove the soundness of our type system under the assumption that any execution of the program is "well-borrowed," as formally stated by using the stacked borrows aliasing model [32], which we adapted for our purposes. Thus, our soundness statement and proofs do not depend on the specific behavior or implementation of the borrow checker. In fact, while Rust's borrow checker is in active development and relatively unstable, our design can account for existing borrow checker variants, including two-phase borrows [12], non-lexical lifetimes [13], and future improvements such as Polonius [58].
- We implement Thrust, an automated modular verification tool for Rust programs based on the prophecy-based refinement type system and CHC-based type inference, as a plugin for the Rust compiler targeting the Rust intermediate language.
- We evaluate Thrust on RustHorn benchmarks and new, additional benchmarks, including challenging cases that go beyond the capabilities of RustHorn and other semi-automated tools. Our results demonstrate Thrust's advantages in both automation and precision compared to state-of-the-art Rust verifiers, including RustHorn, Flux, and Prusti.

1.3 Organization of the Paper

The rest of the paper is organized as follows. Section 2 provides an overview of Thrust and its design philosophy using concrete examples. Sections 3 and 4 respectively formalize our target core language of Rust and the prophecy-based refinement type system. Section 5 describes the implementation of Thrust and reports on the experimental evaluation. Section 6 discusses related work, and Section 7 concludes the paper with remarks on future work. We omit some parts of the formalization. The full definitions and proofs are found in the supplementary material [51].

Fig. 1. A dependent refinement type

```
1 // (x : i32) → {v : i32 | v = x + 1}
2 #[thrust::requires(true)]
3 #[thrust::ensures(result == x + 1)]
4 fn incr(x: i32) -> i32 {
5   x + 1
6 }
```

Fig. 2. Dynamic selection of mutable references

```
1 // (p : &mut i32) → (q : &mut i32) →
2 // { v : () | *p ≥ *q ∧ ○p = *p + 1 ∧ ○q = *q
2 //          ∨ *p < *q ∧ ○p = *p ∧ ○q = *q + 1 }
3 fn incr_max(p: &mut i32, q: &mut i32) {
4   let r = if *p >= *q { p } else { q };
5   *r += 1;
6 }
```

Fig. 3. Refinement typing using prophecy in cases involving borrowing by Thrust

```
1 fn main() {
2   let mut x = Box::new(2);
3   // x : ⟨2⟩
4   let m = &mut *x; // prophecy l
5   // x : ⟨l⟩, m : ⟨2, l⟩
6   decr(m); // assume l = 2 − 1
7   assert!(*x == 1);
8 }
```

```
1 // (x : &mut i32) → {v : () | ○x = *x − 1}
2 fn decr(x: &mut i32) {
3   // initial value x0, prophecy l
4   // x : ⟨x0, l⟩
5   x -= 1;
6   // x : ⟨x0 − 1, l⟩
7   // assume ℓ = x0 − 1
8 }
```

2 Overview

This section provides a high-level description of Thrust's refinement type system, particularly explaining, with examples, how the notion of prophecy, following RustHorn's approach, is used in Thrust to verify programs by refinement types.

In refinement type systems, base types are qualified with logical predicates for refinements. For example, the dependent refinement type of the `incr` function in Fig. 1 states that the function, if it terminates, returns an integer that is the argument `x` plus 1. In this way, refinement predicates constrain the possible values of base types, enabling pre- and post-conditions to be specified. The explicitly written logical predicate in this example can be automatically inferred using CHC-based refinement type inference techniques, ensuring consistency with requirements from other functions and assertions.

However, it is well-known that assigning precise refinement types becomes challenging in cases involving pointers with mutability and aliasing. To address this, previous type systems typically track aliasing situations, allowing for strong updates to pointer destinations when aliasing is guaranteed to be absent and weak updates otherwise. Various methods have been proposed to ensure the absence of aliasing: one approach is to distinguish between concrete locations (where aliasing can be guaranteed to be absent) and abstract locations, as used in [53], while another approach, as seen in ConSORT [60], uses fractional permissions/ownership to enforce non-aliasing. In ConSORT, a method that leverages must-alias user annotations is also provided to enable flexible ownership transfer. Returning to verification for Rust, its type system already guarantees "aliasing XOR mutability" through ownership. Thus, this feature is key to achieving precise verification in refinement type systems for Rust.

For example, Fig. 2 demonstrates how our Thrust assigns precise refinement types to pointer-manipulating programs through strong updates. The `incr_max` function takes two mutable references, `p` and `q`, and the Rust type system guarantees that they do not alias. Inside the function, only the referent of the larger of `p` or `q` is incremented. Thrust can precisely represent this effect

in the refinement predicate of the return type, as shown on line 2, and can also automatically synthesize the predicate through type inference. There, $\circ p$ and $\circ q$[2] respectively refer to the values of the referents of the mutable references `p` and `q` at the time the function returns (when `p` and `q` are dropped), while $*p$ and $*q$ represent the values of their referents at the time of the function call. It is worth mentioning that Flux [43], which leverages exclusive ownership types to allow strong updates via strong references (a variant of mutable references introduced by Flux), cannot assign precise types to programs that dynamically select the target for writing through mutable references based on runtime values, as in this example (and the `take_max` example in [47], where the selected mutable reference `r` is returned to the call site and further updated). Furthermore, it also cannot automatically infer such types.

Figure 3 illustrates how Thrust utilizes the concept of prophecy in verifying programs that involve borrowing. The mutable borrowing reference `m`, created on line 4, is dropped on line 6, returning ownership to `x`, which allows the referent of `x` to be accessed on line 7. However, because `m` is passed to the `decr` function and its referent is updated, it is necessary to correctly propagate this information back to the original owner, `x`. To this end, inspired by RustHorn, Thrust represents the mutable reference `x`, the formal argument of `decr`, as a pair of logical variables, denoted by $\langle x_0, \ell \rangle$, where x_0 represents the initial value of the referent (that is, $*x$) and ℓ is a *prophecy variable* that represents the final value of the referent at the end of the lifetime of the mutable reference `x` (that is, ℓ represents $\circ x$). In general, given a logical representation $\langle a, \ell \rangle$ assigned to a mutable reference, the variable a represents the *current value* of the referent of the mutable reference. Because the current value at the time of the function call is the initial value, x_0 corresponds to $*x$. Thrust reasons about the change on mutable references, as well as owned pointers as shown shortly, by updating the logical representations assigned to them in a flow-sensitive manner. In `decr`, the mutable reference `x` is decremented on line 4. When a mutable reference is updated, the current value in its logical representation is also updated using the assigned value. Therefore, after line 4, the logical representation of `x` is changed to $\langle x_0 - 1, \ell \rangle$, which means that the current value after line 4 is $x_0 - 1$, that is, the initial value minus 1. The value of the prophecy variable ℓ is determined to be the current value at the end of the lifetime of `x`. Because it coincides with the end of the function, Thrust determines the value of ℓ as $x_0 - 1$, the current value just before the function ends. Therefore, in the return type of the function, it is possible to ensure $\circ x = *x - 1$. In the `main` function, the owned pointer `x`, created on line 2, is represented by $\langle 2 \rangle$, where 2 represents the current value of the referent, denoted by $*x$, of `x`. The mutable reference `m` is then created on line 4 by borrowing. Here, the current value $*m$ of `m` is $*x$, which is 2. Since the prophecy $\circ m$ is unknown at this point, a fresh prophecy variable l is assigned. Here, it is important to note that the ownership of `x` has moved to `m`, so the value of $*x$ changes to the value of $\circ m$, which is l. This is because $\circ m$ represents the value of `m` when its lifetime ends, and when ownership returns to `x`, it takes on the value of $\circ m$. The argument `m` of `decr` on line 6 then represents $\langle 2, l \rangle$, and the refinement in the return type of `decr` ensures that $l = 2 - 1$. Thus, we can conclude that the assertion condition on line 7 holds.

Our prophecy-based refinement type system also works well on *reborrowing*, which is crucial for reusing the same mutable reference multiple times under the move semantics of Rust. For example, consider the example in Fig. 4. Type checking proceeds as in the `main` function in Fig. 3, up to line 4. On line 5, the content of the mutable reference `m` is borrowed, that is, reborrowing occurs here. A new prophecy variable l_2 is also introduced for the reborrowing reference `n`. Similarly to how borrowing works with owned pointers, the current value of the mutable reference `m` is

[2]Note the difference of the font style between the variables p, q in the predicate and `p`, `q` in the program.

replaced by the prophecy variable l_2. The mutable reference n is released after line 7, finalizing an assignment that decreases its value. Therefore, at this point, it is ensured that $l_2 = 2 - 1$. On line 8, m's value is decreased by 1 and m is released at the same time. Finally, because x's content is assumed to be equivalent to l_1, we can conclude that the assertion checking never fails.

Fig. 4. Example of reborrowing

```
1  let mut x = 2;
2  // x : ⟨2⟩
3  let m = &mut *x; // prophecy l1
4  // x : ⟨l1⟩, m : ⟨2, l1⟩
5  let n = &mut *m; // prophecy l2
6  // x : ⟨l1⟩, m : ⟨l2, l1⟩, n : ⟨2, l2⟩
7  *n -= 1; // assume l2 = 2 − 1
8  *m -= 1; // assume l1 = l2 − 1
9  assert!(x == 0);
```

Separation of Program and Implicit Variables for Soundness. Our type system tracks state changes by updating the logical representations assigned to variables. For example, in Fig. 3, the current state of the referent of variable x changes from $\langle 2 \rangle$ to $\langle l \rangle$ via borrowing. However, as is well known, we need to be careful when state transitions are involved in refinement type systems, because allowing variables referenced by refinements to change can easily lead to unsoundness. For example, assume that the main function declares an integer variable y with a refinement type $\{y : \mathtt{i32} \mid y = *x\}$. If the variable y is initialized by integer 2, the refinement of the type of y is valid at the point where the owned pointer x is created. However, after the call to decr, the current value of the referent of x changes to 1 (as inferred using the prophecy variable l). Therefore, the refinement of the type of y becomes invalid, even though the variable y itself is not modified.

Our approach to this problem is to distinguish between *program variables* and *logical variables*. Logical variables represent static values, that is, their meaning does not change even as the program evaluation proceeds. The current value of each program variable can be described using logical variables. For example, consider the decr function in Fig. 3. As explained above, the logical representation of the value held by the formal argument x of decr is $\langle x_0, \ell \rangle$ using the logical variable x_0. The logical representation $\langle x_0, \ell \rangle$ assigned to x can be considered as a snapshot of the value of x at the beginning of decr. If the referent of x is mutated to, e.g., number 42, the representation assigned to x is changed to $\langle 42, \ell \rangle$, which indicates that the current value is irrelevant to the value that x held at the beginning of decr. Our type system allows refinements to depend only on logical variables. Because the meaning of logical variables does not change during program execution, we can ensure the soundness of verification based on our type system. We call logical variables *implicit variables* because they do not appear in programs and are introduced implicitly by the type system. Note that prophecy variables are also implicit variables.

3 λ_{COR}: λ-Calculus with Ownership and Reference

This section introduces λ_{COR}, our formalization of the core fragment of Rust.[3] Due to space restriction, we only define the source syntax of λ_{COR} in this section; a high-level overview of the extension to run-time terms and the semantics is described in Section 4.3, and the full definition is found in the supplementary material [51].

The syntax of λ_{COR} is shown in Fig. 5. We use the metavariables x, y, and z for variables, $\mathfrak{i}$ for integers, and *op* for binary operations on integers. We use the underscore _ for binding unused variables. For simplicity, we assume that all the functions are declared globally and their names are denoted by the metavariables f and g.

Values v in λ_{COR} consist of variables, integers, function names, and tuples of values. Places w are values with zero or more applications of dereference and tuple projection.

[3]The subscription "COR" originates from RustHorn's calculus named Calculus of Ownership and Reference (COR).

$$\textbf{Variables}\ x, y, z \qquad \textbf{Function names}\ f, g$$
$$\textbf{Operations}\ op ::= + \mid \leq \mid \cdots \qquad \textbf{Integers}\ \mathfrak{i} ::= 0 \mid 1 \mid -1 \mid \cdots$$

$$\begin{array}{rrcl}
\textbf{Values} & v & ::= & x \mid \mathfrak{i} \mid f \mid (v_1, \cdots, v_n) \\
\textbf{Places} & w & ::= & v \mid *w \mid w.i \\
\textbf{Mutabilities} & \mathfrak{m} & ::= & \mathsf{mut} \mid \mathsf{immut} \\
\textbf{Expressions} & e & ::= & w \mid v_1\ op\ v_2 \mid v_1\ v_2 \mid \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \mid \mathsf{ifz}\ v\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \mid \mathsf{assertz}\ v \mid \\
& & & \mathsf{box}\ v \mid \&^{\mathfrak{m}}\ w \mid v_1 := v_2 \mid \mathsf{drop}\ v
\end{array}$$

Fig. 5. Syntax of λ_{COR}

Expressions e in λ_{COR} consist of places, standard constructs in functional programming (application of integer operations, function application, let-binding, and conditional branching), assertion checking assertz v, which fails only when the value v is nonzero, and instructions for memory manipulation. An instruction box v creates a new owned pointer whose contents are the value v. An instruction $\&^{\mathfrak{m}}\ w$ creates a reference of the pointer specified by the place w. Here, $\mathfrak{m}$ specifies the mutability of the created reference: if $\mathfrak{m}$ is mut (resp. immut), a mutable (resp. immutable) reference is created. Assignment of a value v_2 to an owned pointer or mutable reference v_1 is carried out by an instruction $v_1 := v_2$. For simplicity, λ_{COR} enforces the release of pointers after assignment. Access—i.e., dereference, assignment, and release—to pointers that have been updated causes a run-time error, and programs with such behavior are excluded from the verification target. This design decision does not lose the expressivity of λ_{COR}: if the user wants to access to an updated pointer, they can first create its mutable reference and update the pointer via the reference. While the reference cannot be accessed after the update, the original pointer can be. For example, given an owned pointer or mutable reference x to integers, the expressions $\mathsf{let}\ \_ = x := 0\ \mathsf{in}\ *x$ and $\mathsf{let}\ \_ = x := 0\ \mathsf{in}\ \mathsf{let}\ \_ = x := 3\ \mathsf{in}\ 0$ result in run-time errors. In contrast, the expressions $\mathsf{let}\ x' = \&^{\mathsf{mut}}\ x\ \mathsf{in}\ \mathsf{let}\ \_ = x' := 0\ \mathsf{in}\ *x$ and $\mathsf{let}\ x' = \&^{\mathsf{mut}}\ x\ \mathsf{in}\ \mathsf{let}\ \_ = x' := 0\ \mathsf{in}\ \mathsf{let}\ \_ = x := 3\ \mathsf{in}\ 0$, rewritten to use a mutable borrowing reference, do not cause errors; both evaluate to 0. Pointers can also be released using drop, which originates from RustHorn. The instruction drop does not explicitly appear in Rust programs, but the support for it enables λ_{COR}'s type system to be independent of a specific borrow checking mechanism. In the implementation, we determine program points to insert drop using a simple live-variable analysis, as described in Section 5.1. Note that, because drop releases pointers, for an owned pointer or mutable reference x to integers, $\mathsf{let}\ \_ = x := 0\ \mathsf{in}\ \mathsf{drop}\ x$ and $\mathsf{let}\ \_ = \mathsf{drop}\ x\ \mathsf{in}\ *x$ cause the run-time error, while $\mathsf{let}\ x' = \&^{\mathsf{mut}}\ x\ \mathsf{in}\ \mathsf{let}\ \_ = x' := 0\ \mathsf{in}\ \mathsf{drop}\ x$ does not. The last two instructions—assignment and drop—plays a crucial role in our prophecy-based refinement type system: the release of a mutable reference enables fixing the value of its prophecy. The type system in Section 4 formalizes this idea.

4 Refinement Type System for λ_{COR}

This section presents a refinement type system for λ_{COR} that guarantees that well-borrowed and well-typed programs never get stuck, which especially indicates that the assertion checking in such a program never fails. We start by introducing the type language of our refinement system in Section 4.1 and then present the type system in Section 4.2. Section 4.3 states soundness of the type system and Section 4.4 discusses extensions that are not formalized but are supported by the implemented tool described in Section 5.1.

4.1 Types

$$
\begin{array}{rrcl}
\textbf{Qualifiers} & q & ::= & \mathsf{own} \mid \&\mathfrak{m} \\
\textbf{Logical Terms} & \mathfrak{t} & ::= & x \mid \mathfrak{i} \mid f \mid \mathfrak{t}_1 \mathit{\ op\ } \mathfrak{t}_2 \mid (\mathfrak{t}_1, \cdots, \mathfrak{t}_n) \mid \mathfrak{t}.i \mid \langle \mathfrak{t} \rangle \mid \langle \mathfrak{t}_1, \mathfrak{t}_2 \rangle \mid \langle\cdot \mathfrak{t} \cdot\rangle \mid *\mathfrak{t} \mid \circ \mathfrak{t} \\
\textbf{Canonical Forms} & \mathfrak{v} & ::= & x \mid \mathfrak{i} \mid f \mid (\mathfrak{v}_1, \cdots, \mathfrak{v}_n) \mid \langle \mathfrak{v} \rangle \mid \langle \mathfrak{v}_1, \mathfrak{v}_2 \rangle \mid \langle\cdot \mathfrak{v} \cdot\rangle \\
\textbf{Sorts} & s & ::= & \mathsf{int} \mid \mathsf{func} \mid s_1 \times \cdots \times s_n \mid \mathsf{own}\ s \mid \mathsf{mut}\ s \mid \mathsf{immut}\ s \\
\textbf{Logical Formulas} & \phi & ::= & \top \mid \neg \phi \mid \phi_1 \wedge \phi_2 \mid \exists x : s.\, \phi \mid \mathfrak{t}_1 == \mathfrak{t}_2 \\
\textbf{Underlying Types} & U & ::= & \mathsf{int} \mid (x : T_1) \to T_2 \mid T_1 \times \cdots \times T_n \mid q\ T \\
\textbf{Types} & T & ::= & \{\, x : U \mid \phi \,\} \\
\textbf{Typing Contexts} & \Gamma & ::= & \emptyset \mid \Gamma, x : T \\
\textbf{Place Contexts} & \Delta & ::= & \emptyset \mid \Delta, x : U \triangleright \mathfrak{v}
\end{array}
$$

Fig. 6. Types.

Figure 6 shows the syntax of type-related notions used in our refinement type system. Underlying types U represent what kinds of values are produced by expressions, including int, dependent function types, product types, and pointer types $q\ T$, where q is a qualifier, which represents that a pointer of the type $q\ T$ is an owned pointer own, a mutable reference &mut, or an immutable reference &immut. A refinement type (or simply type) $\{\, x : U \mid \phi \,\}$ refines the values of an underlying type U with a logical formula ϕ, called a *refinement*, that refers to the values trough a variable x. Our logic to describe refinements is a many-sorted first-order logic equipped with term representations for pointers. The sort func is for function names, and own s, mut s, and immut s are for owned pointers, mutable references, and immutable references, respectively, with contents of the sort s. Formulas in the logic are constructed by standard logical connectives along with equality over terms. Terms $\mathfrak{t}$ are variables, integers, function names, and pointer representations as well as operations over them. Terms $\langle \mathfrak{t} \rangle$ and $\langle\cdot \mathfrak{t} \cdot\rangle$ represent owned pointers and immutable references, respectively, with the current values denoted by $\mathfrak{t}$. Terms of the form $\langle \mathfrak{t}_1, \mathfrak{t}_2 \rangle$ represent mutable references, with $\mathfrak{t}_1$ denoting the current value and $\mathfrak{t}_2$ the future value. Operations $*\mathfrak{t}$ and $\circ\mathfrak{t}$ return the current and future values of a pointer $\mathfrak{t}$, respectively ($\circ\mathfrak{t}$ is valid only when $\mathfrak{t}$ is a mutable reference term). We denote canonical forms of terms by $\mathfrak{v}$.

As explained in Section 2, we separate program and implicit variables. To implement it, our type system introduces two kinds of contexts: *typing contexts* Γ, which keep implicit variables introduced by the type system, and *place contexts* Δ, which keep the types and logical representations assigned to program variables. Typing contexts are sequences of bindings of the form $x : T$ for implicit variables x. Place contexts are sequences of bindings of the form $x : U \triangleright \mathfrak{v}$, which means that the variable x locally bounded in a source program (that is, introduced by a function declaration or let-expression) has the type U with the logical canonical form $\mathfrak{v}$ which may be described using implicit variables. We write $\mathrm{dom}(\Delta)$ and $\mathrm{dom}(\Gamma; \Delta)$ for the set of variables bounded by Δ and that bounded by Γ and Δ, respectively.

For example, consider an expression $\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$ and assume that the expression e_1 has a type $\{\, x' : \&\mathsf{mut}\ T \mid \top \,\}$ where $T = \{\, x'' : \mathsf{int} \mid 0 \le x'' \,\}$.[4] To statically represent the state of the pointer returned by e_1, our type system can introduce implicit variables y_1 and y_2 that stand for the current and future values, respectively, of the returned pointer. The type information about y_1 and y_2 can be described by a typing context that includes $y_1 : T$ and $y_2 : T$. Note that the type T assigned to y_1

[4]The logic in Fig. 6 does not directly support the predicate $\le$, but it can be encoded by assuming that $\mathfrak{t}_1 \le \mathfrak{t}_2$ returns 0 or 1 depending on the comparison result.

Place typing rules for underlying types $\boxed{\Gamma; \Delta \vdash w : U \triangleright \mathfrak{v}}$

$$\frac{\Gamma \vdash \Delta \quad x : U \triangleright \mathfrak{v} \in \Delta}{\Gamma; \Delta \vdash x : U \triangleright \mathfrak{v}}\ \text{TP\_Var} \qquad \frac{\Gamma \vdash \Delta}{\Gamma; \Delta \vdash \mathfrak{i} : \mathsf{int} \triangleright \mathfrak{i}}\ \text{TP\_Int}$$

$$\frac{\Gamma \vdash \Delta \quad f : (x : T_1) \to T_2 \in \Phi}{\Gamma; \Delta \vdash f : (x : T_1) \to T_2 \triangleright f}\ \text{TP\_Fun} \qquad \frac{\Gamma; \Delta \vdash w : U \triangleright \mathfrak{v} \quad \Gamma \vdash U <: U'}{\Gamma; \Delta \vdash w : U' \triangleright \mathfrak{v}}\ \text{TP\_USub}$$

$$\frac{\forall\, i \in [1, n].\ \Gamma; \Delta \vdash v_i : T_i \triangleright \mathfrak{v}_i}{\Gamma; \Delta \vdash (v_1, \cdots, v_2) : T_1 \times \cdots \times T_n \triangleright (\mathfrak{v}_1, \cdots, \mathfrak{v}_n)}\ \text{TP\_Tuple}$$

$$\frac{\Gamma; \Delta \vdash w : T_1 \times \cdots \times T_n \triangleright (\mathfrak{v}_1, \cdots, \mathfrak{v}_n) \quad i \in [1, n]}{\Gamma; \Delta \vdash w.i : T_i.U \triangleright \mathfrak{v}_i}\ \text{TP\_Proj}$$

$$\frac{\Gamma; \Delta \vdash w : q\ T \triangleright \mathfrak{v} \quad \mathfrak{v} \in \{\langle \mathfrak{v}' \rangle, \langle \mathfrak{v}', \mathfrak{v}'' \rangle, \langle \cdot \mathfrak{v}' \cdot \rangle\}}{\Gamma; \Delta \vdash *w : T.U \triangleright \mathfrak{v}'}\ \text{TP\_Deref}$$

Place typing rules for types $\boxed{\Gamma; \Delta \vdash w : T \triangleright \mathfrak{v}}$

$$\frac{\Gamma; \Delta \vdash w : U \triangleright \mathfrak{v} \quad x \notin \mathrm{fv}(\mathfrak{v})}{\Gamma; \Delta \vdash w : \{\, x : U \mid x == \mathfrak{v} \,\} \triangleright \mathfrak{v}}\ \text{TP\_Refine} \qquad \frac{\Gamma; \Delta \vdash w : T \triangleright \mathfrak{v} \quad \Gamma \vdash T <: T'}{\Gamma; \Delta \vdash w : T' \triangleright \mathfrak{v}}\ \text{TP\_TSub}$$

Fig. 7. Place typing.

and y_2 are matched with the type of the contents of the pointer returned by e_1. Using these implicit variables, the place context for typechecking e_2 can be given as $x : \mathsf{own}\ T \triangleright \langle y_1, y_2 \rangle$.

4.2 Type System

Our refinement type system consists of four parts: well-formedness for typing contexts, types, and place contexts; subtyping; typing of place expressions; and typing of expressions. In what follows, we explain the well-formedness, typing, and subtyping in the order.

Well-Formedness. Well-formedness for typing contexts $\vdash \Gamma$, and for types $\Gamma \vdash U$ and $\Gamma \vdash T$, is defined straightforwardly and ensures that all refinements are well-sorted under the typing contexts. Well-formedness for place contexts $\Gamma \vdash \Delta$ checks that, for any $x : U \triangleright \mathfrak{v} \in \Delta$, $\Gamma \vdash U$ holds and the logical term $\mathfrak{v}$ is well-sorted under the typing context Γ.

Typing of Place Expressions. Typing judgments for place expressions take the form $\Gamma; \Delta \vdash w : U \triangleright \mathfrak{v}$ or $\Gamma; \Delta \vdash w : T \triangleright \mathfrak{v}$, which means that a place expression w has a type U or T under a typing context Γ and place context Δ and its evaluation result has a logical form described by $\mathfrak{v}$. The inference rules for these typing judgments are presented in Fig. 7. Here, we adopt the notation $T.U$, which stands for the outermost underlying type of the refinement type T. The rule (TP_Var) means that the type and logical form of a variable is found from the given place context. The rule (TP_Fun) assumes a set Φ of pairs $f : (x : T_1) \to T_2$ of a function name f and its type $(x : T_1) \to T_2$. The rules (TP_USub) and (TP_TSub) rely on subtyping $\Gamma \vdash U <: U'$ and $\Gamma \vdash T <: T'$, which will be explained shortly. The rules (TP_Tuple) and (TP_Proj) are defined straightforwardly. The rule (TP_Deref) returns the logically represented current values of pointers as the result of dereference. The rule (TP_Refine) enables assigning a refinement type comprising the assigned underlying type and the logically represented evaluation result. The metafunction fv returns the set of free variables occurring in a given entity.

Expression typing rules $\boxed{\Gamma_1; \Delta_1 \vdash e : T \dashv \Gamma_2; \Delta_2}$ $\boxed{\Gamma; \Delta \vdash e : T} \stackrel{\text{def}}{=} \Gamma; \Delta \vdash e : T \dashv \Gamma; \Delta$

$$\frac{\Gamma; \Delta \vdash w : T \triangleright \mathfrak{v}}{\Gamma; \Delta \vdash w : T}\ \text{T\_Place} \qquad \frac{\Gamma_1; \Delta_1 \vdash e_1 : T_1 \dashv \Gamma; \Delta \quad y \text{ is fresh} \quad y : T_1 \overset{\theta}{\Longrightarrow} \Gamma' \quad \Gamma, \Gamma'; \Delta, x : T_1.U \triangleright \theta(y) \vdash e_2 : T_2 \dashv \Gamma_2; \Delta_2 \quad x \notin \mathrm{dom}(\Delta_2)}{\Gamma_1; \Delta_1 \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : T_2 \dashv \Gamma_2; \Delta_2}\ \text{T\_Let}$$

$$\frac{\Delta = \Delta_1, z : \mathfrak{U}[T] \triangleright Q[\mathfrak{v}], \Delta_2 \quad \mathfrak{U}^{-1} = *P \quad Q^{-1} = *P \quad \Delta' = \Delta_1, z : \mathfrak{U}[T] \triangleright Q[y], \Delta_2 \quad x \notin \mathrm{fv}(\langle \mathfrak{v}, y \rangle) \quad y \notin \mathrm{dom}(\Gamma; \Delta) \quad \Gamma \vdash \Delta}{\Gamma; \Delta \vdash \&^{\mathsf{mut}}\, P[z] : \{\, x : \&\mathsf{mut}\ T \mid x == \langle \mathfrak{v}, y \rangle \,\} \dashv \Gamma, y : T; \Delta'}\ \text{T\_MutBorrow}$$

$$\frac{\Gamma; \Delta \vdash w : q\ T \triangleright \mathfrak{v}}{\Gamma; \Delta \vdash \&^{\mathsf{immut}}\, w : \{\, \&\mathsf{immut}\ T \mid \langle *\mathfrak{v} \cdot \rangle \,\}}\ \text{T\_ImmutBorrow}$$

$$\frac{\Gamma; \Delta \vdash v : U \triangleright \mathfrak{v} \quad U \in \{\mathsf{own}\ T, \&\mathsf{immut}\ T\}}{\Gamma; \Delta \vdash \mathsf{drop}\ v : \{\, \_ : \mathsf{int} \mid \top \,\}}\ \text{T\_Drop} \qquad \frac{\Gamma; \Delta \vdash v : \&\mathsf{mut}\ T \triangleright \langle \mathfrak{v}_1, \mathfrak{v}_2 \rangle}{\Gamma; \Delta \vdash \mathsf{drop}\ v : \{\, \_ : \mathsf{int} \mid \mathfrak{v}_1 == \mathfrak{v}_2 \,\}}\ \text{T\_DropMut}$$

$$\frac{\Gamma; \Delta \vdash v_1 : \&\mathsf{mut}\ T \triangleright \langle \mathfrak{v}_{11}, \mathfrak{v}_{12} \rangle \quad \Gamma; \Delta \vdash v_2 : T \triangleright \mathfrak{v}_2}{\Gamma; \Delta \vdash v_1 := v_2 : \{\, \_ : \mathsf{int} \mid \mathfrak{v}_{12} == \mathfrak{v}_2 \,\}}\ \text{T\_Assign}$$

$$\frac{\Gamma; \Delta \vdash v : T \triangleright \mathfrak{v} \quad x \notin \mathrm{fv}(\mathfrak{v})}{\Gamma; \Delta \vdash \mathsf{box}\ v : \{\, x : \mathsf{own}\ T \mid x == \langle \mathfrak{v} \rangle \,\}}\ \text{T\_Box} \qquad \frac{\Gamma; \Delta \vdash v : \mathsf{int} \triangleright \mathfrak{v} \quad \Gamma \models \mathfrak{v} == 0}{\Gamma; \Delta \vdash \mathsf{assertz}\ v : \{\, \_ : \mathsf{int} \mid \top \,\}}\ \text{T\_Assertz}$$

Expansion rules $\boxed{\Gamma_1 \overset{\theta}{\Longrightarrow} \Gamma_2}$

$$\frac{}{\emptyset \overset{\emptyset}{\Longrightarrow} \emptyset}\ \text{TE\_Empty} \qquad \frac{\Gamma_1 \overset{\theta}{\Longrightarrow} \Gamma_2}{x : T, \Gamma_1 \overset{[x/x] \uplus \theta}{\Longrightarrow} x : T, \Gamma_2}\ \text{TE\_Nonexp}$$

$$\frac{y \text{ is fresh} \quad y : (T \wedge y.\, \phi[\langle y \rangle / x]), \Gamma_1[\langle y \rangle / x] \overset{[\mathfrak{v}/y] \uplus \theta}{\Longrightarrow} \Gamma_2}{x : \{\, x : \mathsf{own}\ T \mid \phi \,\}, \Gamma_1 \overset{[\langle \mathfrak{v} \rangle / x] \uplus \theta}{\Longrightarrow} \Gamma_2}\ \text{TE\_Own}$$

$$\frac{x_1 \text{ is fresh} \quad x_2 \text{ is fresh} \quad x_1 : (T \wedge x_1.\, \phi[\langle x_1, x_2 \rangle / x]), \Gamma_1[\langle x_1, x_2 \rangle / x] \overset{[\mathfrak{v}_1/x_1] \uplus \theta}{\Longrightarrow} \Gamma_2}{x : \{\, x : \&\mathsf{mut}\ T \mid \phi \,\}, \Gamma_1 \overset{[\langle \mathfrak{v}_1, x_2 \rangle / x] \uplus \theta}{\Longrightarrow} x_2 : T, \Gamma_2}\ \text{TE\_Mut}$$

$$\frac{\begin{array}{c} x_1, \cdots, x_n \text{ are fresh} \\ \Gamma_1' = x_1 : T_1, \cdots, x_{n-1} : T_{n-1}, x_n : (T_n \wedge x_n.\, \phi[(x_1, \cdots, x_n)/x]) \\ \Gamma_1', \Gamma_1[(x_1, \cdots, x_n)/x] \overset{[\mathfrak{v}_1/x_1] \uplus \cdots \uplus [\mathfrak{v}_n/x_n] \uplus \theta}{\Longrightarrow} \Gamma_2 \end{array}}{x : \{\, x : T_1 \times \cdots \times T_n \mid \phi \,\}, \Gamma_1 \overset{[(\mathfrak{v}_1, \cdots, \mathfrak{v}_n)/x] \uplus \theta}{\Longrightarrow} \Gamma_2}\ \text{TE\_Tuple}$$

Fig. 8. Expression typing (excerpt).

Typing of Expressions. Typing judgments for expressions take the form $\Gamma_1; \Delta_1 \vdash e : T \dashv \Gamma_2; \Delta_2$, which means that under a typing context Γ_1 and place context Δ_1, an expression e has a type T that is well-formed under Γ_2 possibly with additional implicit variables, and the execution of e changes

the state of pointers described by Δ_1 into that by Δ_2. We write $\Gamma_1; \Delta_1 \vdash e : T$ if $\Gamma_1 = \Gamma_2$ and $\Delta_1 = \Delta_2$. Figure 8 only presents key typing rules for expressions; the full set of the rules is found in the supplementary material [51].

We first explain the typing rules that are easier to understand. The rule (T_Place) simply relies on the typing for place expressions to typecheck them. The rule (T_ImmutBorrow) is for immutable borrowing, stating that the contents of a created immutable reference is borrowed from the pointer specified by a given place expression. The rule (T_Drop) for dropping owned pointers and immutable references guarantees nothing more than simple type safety. By contrast, (T_DropMut) means that, when a mutable reference v is dropped, the interpretation of its future value that is logically represented by $\mathfrak{v}_2$, is fixed to the current value of v that is logically represented by $\mathfrak{v}_1$. The rule (T_Assign) for assignment is similar except that the future value is fixed to the assigned value. The rule (T_Box) means that the instruction box creates a new owned pointer with the specified contents, and (T_Assert) requires that the typing context imply that the logical representation $\mathfrak{v}$ of the argument of assertz is zero. This requirement is formulated using the validity checking $\Gamma \models \mathfrak{v} == 0$ introduced shortly.

The remaining typing rules are (T_Let) and (T_MutBorrow), which are the most complex typing rules in our type system. The complexities originate from the facts that let expressions introduces implicit variables (T_Let) and that creating mutable references introduces prophecy variables (T_MutBorrow).

To explain (T_Let), consider a let expression $\mathsf{let}\, x = e_1 \,\mathsf{in}\, e_2$ under a given typing context Γ_1 and place context Δ_1. Then, (T_Let) requires that e_1 is typechecked under Γ_1 and Δ_1, which means that $\Gamma_1; \Delta_1 \vdash e_1 : T_1 \dashv \Gamma; \Delta$ needs to holds for some T_1, Γ, and Δ. The "post" typing context Γ and place context Δ are used to typecheck the post expression e_2. Our type system allows describing the shape of the value of e_1 in a logical form using implicit variables. What shape is given is determined by an *expansion judgment* $y : T_1 \overset{\theta}{\Longrightarrow} \Gamma'$. This judgment means that an implicit variable y of type T_1 is expanded using newly introduced implicit variables in Γ'. The inference rules for expansion judgments are shown in the bottom of Fig. 8. It should be noted that prophecy variables are introduced in (TE_Mut) to represent the future values of mutable references. Given a refinement type $T = \{ x : U \mid \phi \}$, the notation $T \wedge x.\phi'$ denotes the refinement type $\{ x : U \mid \phi \wedge \phi' \}$. The metavariable θ stands for a mapping from implicit variables to logical canonical forms. In the expansion judgment, θ describes the logical form $\theta(y)$ into which y is expanded. For example, if $T_1 = \{ y : \&\mathsf{mut}\, \{ z_1 : \mathsf{int} \mid \phi_1 \} \mid \phi_2 \}$, we can derive $y : T_1 \overset{\{y \mapsto \langle z_1, z_2 \rangle\}}{\Longrightarrow} z_2 : \{ z_1 : \mathsf{int} \mid \phi_1 \}, z_1 : \{ z_1 : \mathsf{int} \mid \phi_1 \wedge \phi_2[\langle z_1, z_2 \rangle / y] \}$, which means that y is expanded to $\langle z_1, z_2 \rangle$ with an implicit variable z_1 and prophecy variable z_2. The type of the variable z_1 inherits the refinement ϕ_2 of the type T_1 to preserve as much information as possible. In (T_Let), the expression e_2 is typechecked under the typing context augmented with Γ' and the place context that assigns to x its logical form $\theta(y)$. We require the post place context Δ_2 not to include the binding for x to avoid scope extrusion; the type system implements the removing of some bindings as subtyping (see the supplementary material for the details).

The rule (T_MutBorrow) typechecks instructions to create mutable borrows. This rule formalizes the idea that, when a mutable reference is created from a pointer, the value the reference holds when it is released becomes the current value of the original pointer. We use prophecy variables to bridge between "the future value of the mutable reference" and "the current value of the pointer in the future time" (specifically, the time when the reference is released). To enable this idea, we need to identify which part of the pointer is borrowed and potentially updated in the future. We implement it using four kinds of contexts: place expression contexts P, type contexts $\mathfrak{T}$ and $\mathfrak{U}$, and

term contexts Q, which are defined as follows:

$$\begin{array}{lcl} P & ::= & [\,] \mid *P \mid P.i \\ \mathfrak{T} & ::= & [\,] \mid \{x : \mathfrak{U} \mid \phi\} \\ \mathfrak{U} & ::= & \mathsf{own}\,\mathfrak{T} \mid \&\mathsf{mut}\,\mathfrak{T} \mid T_1 \times \cdots \times T_{i-1} \times \mathfrak{T} \times T_{i+1} \times \cdots \times T_n \\ Q & ::= & [\,] \mid \langle Q \rangle \mid \langle Q, \mathfrak{t}_2 \rangle \mid (\mathfrak{t}_1, \cdots, \mathfrak{t}_{i-1}, Q, \mathfrak{t}_{i+1}, \cdots, \mathfrak{t}_n) \end{array}$$

A place expression context P produces a place expression $P[v]$ when its hole is filled with a value v. A type context $\mathfrak{T}$ and $\mathfrak{U}$ produce a type $\mathfrak{T}[T]$ and underlying type $\mathfrak{U}[T]$ when their holes are filled with a type T, respectively. Assume that a variable z has an underlying type U and logical form $\mathfrak{v}$. When a place expression $P[z]$ is borrowed, the type and logical representation of the borrowed part can be identify by P, U, and $\mathfrak{v}$. For example, if $P = (\,[\,].2)$, $U = T_1 \times \{\, x : \mathsf{own}\, T_2 \mid \phi \,\}$, and $\mathfrak{v} = (x_1, \langle x_2 \rangle)$, the type and logical form of the borrowed value (that is the contents of the pointers specified by $P[z]$) are T_2 and x_2, respectively. This extraction of the information about the borrowing can be implemented by viewing place expression contexts as the *inverse* of type and term contexts. Here, we use the notation $P_1[P_2]$ that denotes the place expression context obtained by filling the hole of P_1 with P_2. The inverse $\mathfrak{T}^{-1}$ and $\mathfrak{U}^{-1}$ of type contexts $\mathfrak{T}$ and $\mathfrak{U}$, respectively, are place expression contexts defined as follows:

$$\begin{array}{c} [\,]^{-1} \stackrel{\text{def}}{=} [\,] \quad \{x : \mathfrak{U} \mid \phi\}^{-1} \stackrel{\text{def}}{=} \mathfrak{U}^{-1} \\ (\mathsf{own}\,\mathfrak{T})^{-1} \stackrel{\text{def}}{=} (\mathfrak{T}^{-1})[*[\,]] \quad (\&\mathsf{mut}\,\mathfrak{T})^{-1} \stackrel{\text{def}}{=} (\mathfrak{T}^{-1})[*[\,]] \\ (T_1 \times \cdots \times T_{i-1} \times \mathfrak{T} \times T_{i+1} \times \cdots \times T_n)^{-1} \stackrel{\text{def}}{=} (\mathfrak{T}^{-1})[[\,].i] \end{array}$$

The inverse Q^{-1} of a term context Q is defined as follows:

$$\begin{array}{c} [\,]^{-1} \stackrel{\text{def}}{=} [\,] \qquad \langle Q \rangle^{-1} \stackrel{\text{def}}{=} (Q^{-1})[*[\,]] \qquad \langle Q, \mathfrak{t}_2 \rangle^{-1} \stackrel{\text{def}}{=} (Q^{-1})[*[\,]] \\ (\mathfrak{t}_1, \cdots, \mathfrak{t}_{i-1}, Q, \mathfrak{t}_{i+1}, \cdots, \mathfrak{t}_n)^{-1} \stackrel{\text{def}}{=} (Q^{-1})[[\,].i] \end{array}$$

For instance, in the above example ($P = (\,[\,].2)$, $U = T_1 \times \{\, x : \mathsf{own}\, T_2 \mid \phi \,\}$, $\mathfrak{v} = (x_1, \langle x_2 \rangle)$), the inverses of the context $\mathfrak{U} = T_1 \times \{x : \mathsf{own}\,[\,] \mid \phi\}$ and $Q = (x_1, \langle\, [\,] \,\rangle)$ are the place context $*P$, which specifies the part to be borrowed. By pattern matching $U = \mathfrak{U}[T_0]$ and $\mathfrak{v} = Q[\mathfrak{v}_0]$ with metavariables T_0 and $\mathfrak{v}_0$, we can represent the type T_2 and logical form x_2 of the part $*P[v]$ to be borrowed by T_0 and $\mathfrak{v}_0$, respectively. Using this idea, given a place expression $P[z]$, (T_MutBorrow) extracts the type T and logical form v of the borrowed part specified by $*P[z]$ from a pre place context Δ. In the post place context Δ', the contents of the pointer is replaced by a newly introduced prophecy variable y because it may be updated by the created mutable reference. The refinement type in the conclusion expresses that its current value comes from the original pointer and its future value is represented by the prophecy variable y.

Subtyping. Most of the inference rules for subtyping judgments $\Gamma \vdash T_1 <: T_2$ and $\Gamma \vdash U_1 <: U_2$ are fairly standard. As in standard refinement type systems [52, 64], the subtyping for refinement types checks that the refinements of a typing context and a subtype imply the refinement of a supertype (S_Refine). The implication checking relies on the interpretation of implicit variables. A judgment $\theta \models \Gamma$ means that the interpretations assigned by θ satisfy the refinements specified by Γ. It relies on the formula $\theta \models \phi$, which means that the formula ϕ holds under the valuation θ, and $\mathsf{formula}(T, x)$, which generates a logical formula stating that x satisfies all the refinements in the type T. A judgment $\Gamma \models \phi$ expresses that the formula ϕ holds under any valuation that satisfies the assumptions specified by the typing context Γ.

Validity rules $\boxed{\theta \models \Gamma}$ $\boxed{\Gamma \models \phi}$

$$\frac{\mathrm{dom}(\theta) = \mathrm{dom}(\Gamma) \quad \forall\, x : T \in \Gamma.\ \theta \models \mathsf{formula}(T, x)}{\theta \models \Gamma}\ \textsc{Valuation} \qquad \frac{\forall\, \theta.\ \theta \models \Gamma \Longrightarrow \theta \models \phi}{\Gamma \models \phi}\ \textsc{Valid}$$

Subtyping rules $\boxed{\Gamma \vdash T_1 <: T_2}$ $\boxed{\Gamma \vdash U_1 <: U_2}$

$$\frac{\Gamma \vdash \{\, x : U_1 \mid \phi_1 \,\} \quad \Gamma \vdash \{\, x : U_2 \mid \phi_2 \,\} \quad \Gamma \vdash U_1 <: U_2 \quad \Gamma, x : \{\, x : U_1 \mid \phi_1 \,\} \models \phi_2}{\Gamma \vdash \{\, x : U_1 \mid \phi_1 \,\} <: \{\, x : U_2 \mid \phi_2 \,\}}\ \text{S\_Refine}$$

$$\frac{\vdash \Gamma}{\Gamma \vdash \mathsf{int} <: \mathsf{int}}\ \text{S\_Int} \qquad \frac{\Gamma \vdash T_{21} <: T_{11} \quad \Gamma, x : T_{21} \vdash T_{12} <: T_{22}}{\Gamma \vdash (x : T_{11}) \to T_{12} <: (x : T_{21}) \to T_{22}}\ \text{S\_Fun}$$

$$\frac{\Gamma \vdash T_1 <: T_2 \quad \Gamma \vdash T_2 <: T_1}{\Gamma \vdash \mathsf{own}\ T_1 <: \mathsf{own}\ T_2}\ \text{S\_Own} \qquad \frac{\Gamma \vdash T_1 <: T_2 \quad \Gamma \vdash T_2 <: T_1}{\Gamma \vdash \&\mathsf{mut}\ T_1 <: \&\mathsf{mut}\ T_2}\ \text{S\_Mut}$$

$$\frac{\Gamma \vdash T_1 <: T_2}{\Gamma \vdash \&\mathsf{immut}\ T_1 <: \&\mathsf{immut}\ T_2}\ \text{S\_Immut} \qquad \frac{\forall\, i \in [1, n].\ \Gamma \vdash T_{1i} <: T_{2i}}{\Gamma \vdash T_{11} \times \cdots \times T_{1n} <: T_{21} \times \cdots \times T_{2n}}\ \text{S\_Tuple}$$

Fig. 9. Subtyping.

Example. As an example of typechecking, consider how the expression

$$\mathsf{let}\ x' = \&^{\mathsf{mut}}\, x\ \mathsf{in}\ \mathsf{let}\ \_ = x' := 0\ \mathsf{in}\ \mathsf{let}\ y = *x\ \mathsf{in}\ \mathsf{assertz}\ y$$

is typechecked under $\Gamma_1 = x_1 : \mathsf{int}, x_2 : \mathsf{int}$ and $\Delta_1 = x : \&\mathsf{mut}\ \mathsf{int} \triangleright \langle x_1, x_2 \rangle$ (we may abbreviate a refinement type $\{\, x : U \mid \phi \,\}$ to U if the refinement ϕ is not important). Note that x_1 and x_2 are an implicit and a prophecy variable, respectively, that can be introduced by the expansion in (T_Let). First, $\&^{\mathsf{mut}}\, x$ is typechecked by (T_MutBorrow) as follows:

$$\Gamma_1; \Delta_1 \vdash \&^{\mathsf{mut}}\, x : \{\, x' : \&\mathsf{mut}\ \mathsf{int} \mid x' == \langle x_1, z \rangle \,\} \dashv \Gamma_1, z : \mathsf{int}; x : \&\mathsf{mut}\ \mathsf{int} \triangleright \langle z, x_2 \rangle\ .$$

By (T_Let), typing and place contexts are augmented. Let $\Gamma_2 = \Gamma_1, z : \mathsf{int}, x_2' : \mathsf{int}, x_1' : \{\, x_1' : \mathsf{int} \mid \langle x_1', x_2' \rangle == \langle x_1, z \rangle \,\}$ and $\Delta_2 = x : \&\mathsf{mut}\ \mathsf{int} \triangleright \langle z, x_2 \rangle, x' : \&\mathsf{mut}\ \mathsf{int} \triangleright \langle x_1', x_2' \rangle$, where the variables x_1' and x_2' are an implicit and a prophecy variable, respectively, by expanding the shape of x'. Next, the typechecking of $x' := 0$ is performed as:

$$\frac{\dfrac{\text{(TP\_Var)}}{\Gamma_2; \Delta_2 \vdash x' : \&\mathsf{mut}\ \mathsf{int} \triangleright \langle x_1', x_2' \rangle} \qquad \dfrac{\text{(TP\_Int)}}{\Gamma_2; \Delta_2 \vdash 0 : \mathsf{int} \triangleright 0}}{\Gamma_2; \Delta_2 \vdash x' := 0 : \{\, \_ : \mathsf{int} \mid x_2' == 0 \,\}\ .}\ \text{(T\_Assign)}$$

Under Δ_2, $*x : \mathsf{int} \triangleright z$ is derivable by (TP_Var) and (TP_Deref). Consider the typechecking of assertz y. Again, the typing and place contexts in assertz y is augmented by (T_Let). Let $\Gamma_3 = \Gamma_2, x_0 : \{\, \_ : \mathsf{int} \mid x_2' == 0 \,\}, y' : \{\, y' : \mathsf{int} \mid y' == z \,\}$ and $\Delta_3 = \Delta_2, \_ : \mathsf{int} \triangleright x_0, y : \mathsf{int} \triangleright y'$. Then, assertz y is typechecked as:

$$\frac{\dfrac{}{\Gamma_3; \Delta_3 \vdash y : \mathsf{int} \triangleright y'}\ \text{(TP\_Var)} \qquad \Gamma_3 \models y' == 0}{\Gamma_3; \Delta_3 \vdash \mathsf{assertz}\ y : \{\, \_ : \mathsf{int} \mid \top \,\}}\ \text{(T\_Assert)}$$

The validity checking holds because Γ_3 includes $y' == z$ and $\langle x_1', x_2' \rangle == \langle x_1, z \rangle$ (so $z == x_2'$) and $x_2' == 0$.

4.3 Soundness

This section describes a high-level overview of soundness of the type system. As with Flux, our type system targets *well-borrowed* programs, that is, ones conforming to the "aliasing XOR mutability" discipline of Rust. To incorporate this discipline, the operational semantics of λ_{COR} employs the *Stacked Borrows* aliasing model [32]. More specifically, a pointer in λ_{COR} is a memory location l equipped with tag t generated at run time, taking the form $\mathsf{ptr}(l, t)$. *Heaps* h in λ_{COR} not only associate memory locations with values, but also keep relations between borrowed and borrowing pointers using stack data structures. The use of stacks enables the run-time detection of the violation of the "aliasing XOR mutability" discipline. The semantics of λ_{COR} is defined by an evaluation relation for configurations, which are tuples $\langle h, \sigma, e\rangle$ of a heap h, an *environment* σ, which maps variables to values, and an expression e (that may contain free variables mapped by σ). After one-step reduction, $\langle h, \sigma, e\rangle$ is transformed to another configuration $\langle h', \sigma', e'\rangle$, written as $\langle h, \sigma, e\rangle \longrightarrow \langle h', \sigma', e'\rangle$, or an error happens due to the violation of "aliasing XOR mutability", which is represented as $\langle h, \sigma, e\rangle \longrightarrow \mathsf{BorrowError}$. As described in Section 3, λ_{COR} assumes that an owned pointer or mutable reference that has been mutated or released is not accessed anymore—i.e., dereferenced, mutated, or released. The run-time error BorrowError also happens if this memory discipline is broken.

Our proof of soundness rests on progress and subject reduction [71]. To prove them, we formulate the invariants on heaps emerging during the execution of well-typed programs. For example, one invariant is that, if a mutable reference is alive, there exists a prophecy variable that matches only with the future value of the reference and also with the "current value" of the pointer the reference borrows. Another invariant is that each pointer exclusively owns the value of its referent. Using such invariants, we can show that progress and subject reduction and then type soundness stated as follows:

Theorem 4.1. *Let h be the empty heap, σ be the empty environment, and e be an expression of the form* $\mathsf{let}\, x = e_0 \,\mathsf{in}\, v$. *Assume that every function has the function type specified by Φ. If $\emptyset; \emptyset \vdash e : T \dashv \Gamma; \Delta$ and $\langle h, \sigma, e\rangle \longrightarrow^* \langle h', \sigma', e'\rangle$, then one of the following holds: $e' = v'$ for some v'; $\langle h', \sigma', e'\rangle \longrightarrow \langle h'', \sigma'', e''\rangle$ for some h'', σ'', and e''; or $\langle h', \sigma', e'\rangle \longrightarrow \mathsf{BorrowError}$.*

Because Rust's borrow checker guarantees that BorrowError never happens, we can ensure that (safe) Rust programs well-typed in our refinement type system never get stuck. In particular, the assertion checks involved in them never fail.

4.4 Discussion and Extensions

Algebraic Data Types. λ_{COR} includes tuples, but does not support general ADTs that include recursive types. However, variants can be trivially encoded with tuples, and we believe that it is not difficult to extend λ_{COR} with recursive types. One might worry that we need to be careful of expansion of the logical representations of let-bound variables in the presence of recursive types because they could lead to infinite expansion, but expansion is necessary only up to visibly accessed values and they are at finite depth.

Function Closures. λ_{COR} includes higher-order functions, but these functions must not contain free variables. A workaround to support functions containing free variables is encoding them as closures (tuples of a closed function and part of arguments). However, this is not completely satisfactory because, under this encoding, the type of a function is aware of the number of free variables and their types. Rust can relax the problem by using a mechanism called *Boxed Closure*, which enables abstracting argument and return types of function types. Roughly speaking, this is

achieved by existential quantification through dynamic dispatch. Introducing existential types into λ_{COR} is left for future work.

5 Implementation and Evaluation

We implemented our static Rust verifier Thrust based on the prophecy-based refinement type system presented in Section 4 and a backend CHC solver Spacer [39, 40],[5] as a plugin for the Rust compiler. Following the CHC-based refinement type inference method in [64], Thrust prepares refinement type templates, generates type derivations based on them, and collects subtyping constraints to form a system of CHC constraints, which is then solved by the backend solver Spacer. Spacer is a solver based on the property directed reachability (PDR) approach [11, 20, 62] and has demonstrated top performance in the benchmarks of the CHC competition (CHC-COMP).

In CHC constraint generation, rather than working directly with the source program, we utilize the Mid-level Intermediate Representation (MIR) [16] of Rust programs, specifically an optimized MIR provided by the Rust compiler after desugaring, type checking, borrow checking, and certain optimizations. By using MIR, which abstracts away much of the complexity of the source program through desugaring and optimization, we were able to implement Thrust with less effort compared to working directly with the source Rust programs while maintaining stability against changes in the source code. We discuss the differences between λ_{COR}, introduced in Section 3, and MIR in Appendix A, detailing the implementation specifics. In the remainder of this section, we report the evaluation results of Thrust.

5.1 Experimental Evaluation

We evaluated Thrust on (1) a new benchmark set designed to compare a broader range of verifiers including those that are not limited to automated verification tools, and on (2) the RustHorn benchmark set [47] for comparing automated verifiers. The benchmark programs are available in the supplementary material [51]. All experiments were conducted on a Ubuntu 24.04 cloud computing instance with 32 GB of RAM and 16 logical CPU cores on an AMD EPYC 7R13 Processor, using Spacer, bundled with Z3 (version 4.13.0), as the backend CHC solver for both Thrust and RustHorn. The time limit was set to 180 seconds.

5.2 Evaluation on the New Benchmark Set

We compared Thrust (commit `d868517`) with existing verification tools for Rust, including RustHorn (https://github.com/hopv/rust-horn/ commit `e1b7b78`), Flux (https://github.com/flux-rs/flux commit `bfa3113`), and Prusti (https://github.com/viperproject/prusti-dev/releases/tag/v-2024-03-26-1504), using a newly prepared benchmark set categorized into four groups based on our focus.

- The "prec" category consists of verification problems that require precise reasoning about mutable references. Some programs involve dynamically selected mutable references based on runtime values, and others include splitting of mutable references of tuples. These programs are challenging to verify with methods that do not support prophecy or similar technique.
- The "ho-funs" category consists of problems that involve higher-order functions.
- The "adt" category consists of problems that involve recursive user-defined data types. "1-list-pos" and "2-dict-pos" require invariants common to all data elements of a recursive ADT for successful verification. The other problems in this category require verification of properties related to the structure or specific elements of recursive ADTs.

[5] Because Thrust outputs CHC constraints in the SMT-LIB 2 format, other CHC solvers can also be used as backends.

Table 2. Experiment results on the new benchmark set

	Thrust			RustHorn		Flux			Prusti		
	Annot.	Result	Time(s)	Result	Time(s)	Annot.	Result	Time(s)	Annot.	Result	Time(s)
prec-1-inc-max-plain-safe	0	safe	0.05	safe	0.04	0	unknown	0.07	0	safe	3.77
prec-1-inc-max-plain-unsafe	0	unsafe†	0.05	unsafe	0.04	0	unknown	0.07	0	unknown	3.28
prec-2-bor-rest-safe	0	safe	0.09	safe	0.05	0	abort	N/A	0	safe	3.36
prec-2-bor-rest-unsafe	0	unsafe†	0.05	unsafe	0.05	0	abort	N/A	0	unknown	3.37
prec-3-opt-get-or-insert-safe	0	safe	0.12	abort	N/A	0	abort	N/A	0	abort	N/A
prec-3-opt-get-or-insert-unsafe	0	unsafe†	0.1	abort	N/A	0	abort	N/A	0	abort	N/A
prec-4-borrow-split-safe	0	safe	0.04	abort	N/A	1	safe	0.07	1	abort	N/A
prec-4-borrow-split-unsafe	0	unsafe†	0.04	abort	N/A	1	unknown	0.07	1	abort	N/A
ho-funs-1-app-mut-safe	0	safe	0.04	abort	N/A	0	unknown	0.05	0	abort	N/A
ho-funs-1-app-mut-unsafe	0	unsafe†	0.04	abort	N/A	0	unknown	0.05	0	abort	N/A
ho-funs-2-return-fn-safe	0	safe	0.06	abort	N/A	0	unknown	0.06	0	abort	N/A
ho-funs-2-return-fn-unsafe	0	unsafe†	0.05	abort	N/A	0	unknown	0.06	0	abort	N/A
ho-funs-3-list-fold-safe	0	safe	0.11	abort	N/A	0	unknown	0.06	0	abort	N/A
ho-funs-3-list-fold-unsafe	0	unsafe†	0.07	abort	N/A	0	unknown	0.06	0	abort	N/A
ho-funs-4-opt-map-safe	0	safe	0.12	abort	N/A	0	unknown	0.06	0	abort	N/A
ho-funs-4-opt-map-unsafe	0	unsafe†	0.07	abort	N/A	0	unknown	0.06	0	abort	N/A
adt-1-list-pos-safe	0	safe	0.28	timeout	N/A	2	safe	0.12	5	safe	6.69
adt-1-list-pos-unsafe	0	unsafe†	0.17	unsafe	0.06	2	unknown	0.12	5	unknown	6.58
adt-2-dict-pos-safe	2	safe	0.17	timeout	N/A	2	safe	0.12	5	safe	7.07
adt-2-dict-pos-unsafe	2	unsafe†	0.11	unsafe	0.06	2	unknown	0.12	5	unknown	6.86
adt-3-inv-safe	0	safe	0.09	safe	0.07	5	safe	0.12	2	safe	5.81
adt-3-inv-unsafe	0	unsafe†	0.07	unsafe	0.06	5	unknown	0.12	2	unknown	5.79
adt-4-dict-get-or-insert-safe	0	safe	4.09	abort	N/A	7	unknown	0.11	5	abort	N/A
adt-4-dict-get-or-insert-unsafe	0	unsafe†	2.72	abort	N/A	7	unknown	0.11	5	abort	N/A
mod-1-sum2-base-safe	0	safe	0.08	safe	0.07	1	safe	0.08	1	safe	4.16
mod-1-sum2-base-unsafe	0	unsafe†	0.08	unsafe	0.05	1	unknown	0.09	1	unknown	4.3
mod-2-sum2-same-safe	1	safe	0.05	abort	N/A	1	safe	0.09	1	safe	4.22
mod-2-sum2-same-unsafe	1	unsafe†	0.04	unsafe	0.05	1	unknown	0.09	1	unknown	4.22
mod-3-nonlin-sum-safe	2	safe	0.05	timeout	N/A	1	safe	0.09	2	safe	4.09
mod-3-nonlin-sum-unsafe	2	unsafe†	0.05	timeout	N/A	1	unknown	0.09	2	unknown	4.14
mod-4-nonlin-mult-mut-safe	2	safe	0.05	timeout	N/A	1	safe	0.09	2	safe	4.17
mod-4-nonlin-mult-mut-unsafe	2	unsafe†	0.05	unsafe	0.05	1	unknown	0.09	2	unknown	4.24

† We manually confirmed that the program is indeed unsafe when Thrust concludes that the program is unsafe, since we have not formally proved the completeness of Thrust, although it is conjectured to be complete for first-order programs for the same reason that RustHorn is, and we also conjecture completeness for the patterns of higher-order function usage found in the benchmark set.

- The "mod" category consists of problems that are challenging for automated verifiers but can be verified with modular verifiers using the help of annotations. Notably, "3-nonlin-sum" and "04-nonlin-mult" involve nonlinear arithmetic, making automated verification challenging.

The benchmark programs were modified for each tool in a way that does not affect their semantics to make them acceptable as inputs. Specifically, in the "prec" category, functions were inlined in Flux and Prusti inputs in order to avoid the need for annotations.

The experimental results are shown in Table 2. The column "Annot." shows the number of lines used for user-specified invariants in function signatures. The "Annot." column for RustHorn is omitted since it does not allow users to write annotations. The column "Result" indicates the verification outcome provided by the tool, showing whether the tool identifies the program as safe, unsafe, or unknown. Some results indicate an abort or a timeout. The column "Time(s)" represents the elapsed verification time in seconds. Some results from Table 2 illustrate the fundamental design differences and limitations between Thrust and other tools. Below, we will discuss the differences in benchmark results for each tool compared to Thrust, exploring whether these differences are fundamental in terms of their limitations.

The differences between RustHorn and Thrust.

- In terms of precision, RustHorn does not differ from Thrust. There are some cases of abort, but these all seem to be non-essential issues of the implementation.
- RustHorn could not handle programs in the "ho" category. It is a fundamental difference. Thrust (and type-based approaches) handle higher-order functions via type-based abstraction and subtyping of refinement types. The RustHorn paper discusses an approach to supporting higher-order functions by using higher-order CHCs. However, since higher-order CHC constraint solving is typically performed using refinement types, a fundamental extension is ultimately required.
- RustHorn could not verify some programs in the "adt" category. It is a fundamental difference. Thanks to type-based abstraction, Thrust reduces the burden on the backend CHC solver by introducing a predicate variable that enables per-element refinement for the type parameter in polymorphic collection types such as lists. In contrast, RustHorn does not utilize types in its verification process and therefore does not introduce such a predicate variable. As a result, since it is necessary to find solutions using recursive functions on ADTs, only certain CHC solvers like HoIce, which support this, can be used to verify programs that involve ADTs.
- RustHorn could not verify programs in the "mod" category. It is a fundamental difference, as RustHorn does not provide a refinement type system for Rust, and therefore, it also lacks a means to introduce annotations in the target Rust program for modular verification.

The differences between Flux and Thrust.

- Flux could not verify some programs in the "prec" category. It is a fundamental difference. In Flux's type system, strong references are tracked as singleton types, which results in a loss of precision for examples that conditionally select mutable references (e.g., Fig. 2). Among these, "3-opt-get-or-insert" is modeled after the Rust standard library's `Option::get_or_insert` function. This widely used method serves as an example demonstrating the real-world applicability of the prophecy-based, precise verification of Thrust.
- Flux could not handle programs in the "ho" category. We believe it is not a fundamental difference. It supports some usage of function closures.[6] Flux is based on refinement types, and there seems to be no reason why it cannot support higher-order functions.
- Flux could not handle "adt-4-dict-get-or-insert-safe" in the "adt" category. It is a fundamental difference. The program requires the propagation of strong updates through a mutable reference that points to a part of a data structure, and the reference is selected dynamically based on the function argument. Flux loses verification precision when tracking dynamically selected mutable references, as explained above.
- In terms of modularity, there is no difference compared to Thrust.
- Flux required some annotation lines in some programs that Thrust verified without any. It is a fundamental difference. As discussed in Section 1.1, Flux cannot infer mutation effects on function types that take a `&strg` (mutable) reference[7].

The differences between Prusti and Thrust.

- Prusti could not verify "prec-3-opt-get-or-insert" and "prec-4-borrow-split" programs in the "prec" category. We consider these not to be fundamental limitations of Prusti's approach, but rather issues that could be resolved with sufficient engineering effort. Aborted cases of "prec-3-opt-get-or-insert" are likely due to Prusti's lack of support for enum references. The Prusti documentation[8] also states that "References in enums are not yet supported."

[6] https://github.com/flux-rs/flux/pull/386
[7] `&strg` is a Flux-specific construct that creates a mutable reference, allowing strong updates on it.
[8] https://viperproject.github.io/prusti-dev/user-guide/tour/pledges.html

Prusti also lacks support for borrow splitting in its current implementation and fails to handle programs such as the "prec-4-borrow-split" cases. We believe that once these issues of enum references and borrow splitting are resolved, there will be no reason these programs cannot be verified with Prusti.

- Prusti could not handle programs in the "ho" category. We consider that it is not a fundamental difference. Prusti had an implementation to support some usage of function closures.[9] The implementation appears to still be in the early stages, but we believe that the extension of Prusti presented in [70] would allow Prusti to support higher-order functions, at least to the same extent that Thrust currently does. Furthermore, another extension of Prusti for reasoning about iterators and closures has also been proposed [8].
- Prusti could not handle "adt-4-dict-get-or-insert" programs in the "adt" category. It is not a fundamental difference. Just like the prec category, aborted cases with these programs are likely due to Prusti's lack of support for enum references.
- Regarding modularity, there is no difference compared to Thrust.
- Prusti required some annotation lines in programs that Thrust verified without any. It is a fundamental difference. As discussed in Section 1.1, Prusti's automation is based on SMT solving, and it requires user annotations of inductive invariants for loops and recursive functions to generate verification conditions.

5.3 Evaluation on the RustHorn Benchmark Set

We compared Thrust's performance (commit `d868517`) with that of another automated verifier, RustHorn (commit `e1b7b78`), using the RustHorn benchmark set. For some of the instances, we made minor modifications that did not affect their semantics to adapt them as inputs for Thrust. In this experiment, Spacer was configured with the `fp.validate=true` flag. The results are summarized in the scatter plot in Fig. 10 and the full results are found in Appendix B. In the scatter plot, safe instances are shown as blue circles, unsafe instances as red circles, and instances where either tool aborted are shown as magenta triangles.

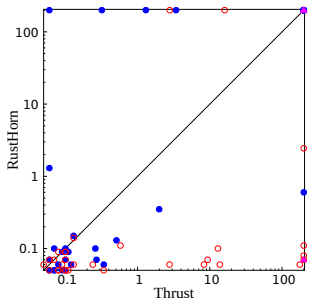


Fig. 10. Summary of experiment results on the RustHorn benchmark set

In most cases, verification with Thrust was completed in a comparable time to RustHorn but was sometimes slower. This is because Thrust interprets MIR more rigorously compared to RustHorn, resulting in more complex constraints being generated. As explained in Section 2, our type system introduces implicit variables at the time of variable binding for modularity (type-system style formalization) while maintaining soundness. The operands of MIR are limited to some simple forms of values, which sometimes results in repeated variable assignments appearing before a terminator in the basic block. Since Thrust interprets MIR statements one by one, this kind of basic block generates many implicit variables within the Thrust environment, leading to an increased load on the backend CHC solver as the number of arguments of predicate variables increases. On the other hand, RustHorn does not have a notion of implicit variables. Moreover, RustHorn performs a kind of symbolic execution to analyze the final environment of the block and generate CHCs based on that, processing one basic block at a time, which results in the generation of simpler

[9] https://github.com/viperproject/prusti-dev/pull/138

CHCs. Additionally, RustHorn aligns the source MIR program with CHC model by interpreting the low-level representation of enum pattern matching within MIR, based on the assumption of a fixed program structure generated by rustc. By contrast, because Thrust interprets MIR as is, the representation of enums becomes more cumbersome[10], which may impact performance. Thus, if Thrust is modified to interpret basic blocks in the same manner as RustHorn by inlining program variables within the block, and if unnecessary predicate variables introduced due to the simplicity of the implementation are eliminated, we expect Thrust to generate the same CHCs as RustHorn, except for programs involving enums. On the other hand, regarding the aforementioned issue with enum representation, we do not intend to implement ad-hoc processes like RustHorn, as we aim to maintain stability against future changes in rustc's MIR generation.

6 Related Work

Since static verifiers for Rust have already been compared in Section 1, this section will discuss an alternative approach to Rust verification, verification techniques for pointer-manipulating programs in languages other than Rust, and related research on the techniques used by Thrust.

There has been research on bug-finding tools that apply *bounded* model checking to Rust [5, 44, 59, 68]. When it comes to automated verification of languages other than Rust, SeaHorn [28] is an automated verifier for C programs based on CHC solving, which encodes the heap as arrays. However, in verification of examples discussed in this paper, synthesis of *quantified* invariants is often required, making it difficult to scale. JayHorn [36], on the other hand, is an automated verifier for Java programs based on CHC solving, where heap invariants are represented by predicate variables. This heap encoding is more efficient than the array encoding, but it comes with lower precision. Additionally, there have been proposals to extend SMT and CHC solvers with background theories that can reason about the heap [19, 21].

Dependent refinement types have been extended to reason about mutable cells. Zhu and Xi [74] introduced a notion of *stateful views* to support pointer manipulation, but their system requires manual annotations to track ownership. Rondon et al. [53] and Bakst and Jhala [4] applied liquid type inference to pointer-manipulating C programs, achieving local strong updates through a technique similar to alias types [56]. The former in particular forms the basis for Flux [43]. Gordon et al. [27] introduced a notion of *rely-guarantee references* to enable local strong updates, unifying ideas from reference immutability type systems and rely-guarantee program logics. Toman et al. [60] presented an automated verification tool ConSORT based on a combination of automatic inference of fractional permission/ownership [10] and CHC-based refinement type inference. However, fractional permission/ownership is different from the Rust-style ownership based on lifetimes and borrowing, and thus cannot be directly applied to Rust verification. There have been attempts to unify the two ownership styles [49], but limitations such as lack of support for nested references remain. RefinedC [54] is an ownership refinement type system for C program verification and forms the basis for RefinedRust [25], but it differs from Thrust in its design philosophy, as it is not an automated verifier but aims to provide foundational proofs based on Iris logic.

The notion of prophecy was introduced in verification of concurrent programs to predict the order of nondeterministic events [1], but it has also been applied to verification of hyperproperties among multiple programs [7, 67], where it is used to predict the behavior of the other programs from a program. As in Thrust, prophecy has been used in Rust verification to propagate update information of mutable borrows to the original owner [17, 25, 47]. Additionally, prophecy is used to predict a continuation effect of call/cc [26], and to enable invariant synthesis that would

[10]We model enums in place contexts explicitly, along with an implicit variable for the enum discriminant. A detailed explanation of our enum encoding is provided in the supplementary material [51].

typically require quantifiers and arrays to be conducted without them [45]. Furthermore, prophecy was integrated into separation logic [35], particularly in Iris. We believe that investigating the application of our prophecy-based refinement type system beyond Rust verification to these domains could be an interesting direction for future research.

7 Conclusion

We presented Thrust, an automated, modular, and precise verification tool for Rust, based on a novel prophecy-based refinement type system and CHC-based refinement type inference techniques. Unlike existing refinement type systems such as low-level Liquid Types, Flux, and ConSORT, which enable strong updates through concrete locations or fractional ownership types, our system enables strong updates of mutable references using prophecy, by propagating the update information of mutable borrows to the original owners through prophecy variables. We have demonstrated that the prophecy-based approach can effectively integrate with type-system-style formalization and fully automated refinement type inference, both theoretically and empirically. We proved the soundness of the type system under the assumption of the well-borrowedness of the program in the sense of stacked borrows, by introducing a core language λ_{COR} of Rust. Thanks to our type-based formulation, Thrust naturally supports higher-order functions and algebraic data types. We also evaluated Thrust and obtained promising results with respect to automation, modularity, and precision, compared to state-of-the-art Rust verifiers including RustHorn, Flux, and Prusti.

Future work includes extending our prophecy-based refinement type system to verify and falsify temporal and branching-time safety and liveness properties. To this end, we consider applying the technique used in this paper for deriving a prophecy-based refinement type system for Rust from an ordinary refinement type system for ML-like functional programs to existing systems for temporal and branching-time safety and liveness verification and falsification [50, 65]. Regarding language features, while it is not difficult to support internal mutability and `unsafe` code through imprecise reasoning with weak updates, developing a mechanism for precise and automated verification of these practically significant features, which offer Rust flexibility but do not benefit from well-borrowedness, remains an important challenge. We are working toward scaling Thrust for verifying real-world Rust programs. If we limit our focus to verifying "safe" Rust programs that use standard library components, without verifying the standard library itself, which contains `unsafe` code, there do not appear to be any fundamental difficulties. The main challenge lies instead in the engineering effort required to support currently unsupported features, such as traits and modules. To support standard library types, we only need to provide a mechanism for attaching refinement types to existing Rust types, commonly referred to as "extern spec." Regarding traits, we only need to support checking trait implementations against the type annotations in the trait definitions, so this does not pose a significant challenge.

Data Availability Statement

Thrust source code can be found at https://github.com/coord-e/thrust. We also provide an artifact [51] to reproduce the evaluation in Section 5.1, including a snapshot of Thrust, related software, and benchmarks.

Acknowledgments

We thank the anonymous reviewers for their useful comments, which improved the presentation of the paper. We would also like to thank the Prusti team, especially Alexander Summers, for kindly responding to our questions regarding Prusti. This research was partially supported by JSPS KAKENHI Grant Numbers JP20H04162, JP20H05703, JP20H00582, JP22H03564, JP23K24826, JP24H00699, and JP25H00446, as well as JST CREST Grant Number JPMJCR21M3.

References

[1] M. Abadi and L. Lamport. 1988. The existence of refinement mappings. In *LICS '88*. 165–175.
[2] Andrew W. Appel. 1998. SSA is functional programming. *ACM SIGPLAN Notices* 33, 4 (1998), 17–20.
[3] Vytautas Astrauskas, Peter Müller, Federico Poli, and Alexander J. Summers. 2019. Leveraging Rust types for modular specification and verification. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 147:1–147:30.
[4] Alexander Bakst and Ranjit Jhala. 2016. Predicate Abstraction for Linked Data Structures. In *VMCAI '16* (St. Petersburg, FL, USA) *(LNCS)*. Springer, 65–84.
[5] Marek Baranowski, Shaobo He, and Zvonimir Rakamarić. 2018. Verifying Rust Programs with SMACK. In *VMCAI '18*. Springer, 528–535.
[6] Jesper Bengtson, Karthikeyan Bhargavan, Cédric Fournet, Andrew D. Gordon, and Sergio Maffeis. 2011. Refinement Types for Secure Implementations. *ACM Transactions on Programming Languages and Systems* 33, 2, Article 8 (Feb. 2011), 45 pages.
[7] R. Beutner and B. Finkbeiner. 2022. Prophecy Variables for Hyperproperty Verification. In *CSF '22*. IEEE, 471–485.
[8] A. Bílý, J. Hansen, P. Müller, and A. J. Summers. 2023. Compositional Reasoning about Advanced Iterator Patterns in Rust. In *International Workshop on Aliasing, Confinement and Ownership in object-oriented programming (IWACO)*.
[9] Nikolaj Bjørner, Arie Gurfinkel, Kenneth L. McMillan, and Andrey Rybalchenko. 2015. Horn Clause Solvers for Program Verification. In *Fields of Logic and Computation II: Essays Dedicated to Yuri Gurevich on the Occasion of His 75th Birthday (LNCS, Vol. 9300)*. Springer, 24–51.
[10] John Boyland. 2003. Checking Interference with Fractional Permissions. In *SAS '03*. Springer, 55–72.
[11] Aaron R. Bradley. 2011. SAT-based Model Checking Without Unrolling. In *VMCAI '11* (Austin, TX, USA) *(LNCS, Vol. 6538)*. Springer, 70–87.
[12] Rust Community. 2017. *2025-nested-method-calls - The Rust RFC Book*. https://rust-lang.github.io/rfcs/2025-nested-method-calls.html.
[13] Rust Community. 2017. *2094-nll - The Rust RFC Book*. https://rust-lang.github.io/rfcs/2094-nll.html.
[14] Rust Community. 2023. *Type coercions - The Rust Reference*. https://doc.rust-lang.org/reference/type-coercions.html.
[15] Rust Community. 2024. *Rust Programming Language*. https://www.rust-lang.org/.
[16] Rust Community. 2024. *The MIR (Mid-level IR) - Rust Compiler Development Guide*. https://rustc-dev-guide.rust-lang.org/mir/index.html.
[17] Xavier Denis, Jacques-Henri Jourdan, and Claude Marché. 2022. Creusot: A Foundry for the Deductive Verification of Rust Programs. In *ICFEM '22 (LNCS, Vol. 13478)*. Springer, 90–105.
[18] The Rust Project Developers. 2019. *Rust Case Study: Community Makes Rust an Easy Choice for npm*. https://www.rust-lang.org/static/pdfs/Rust-npm-Whitepaper.pdf.
[19] Gregory J. Duck, Joxan Jaffar, and Nicolas C. H. Koh. 2013. Constraint-Based Program Reasoning with Heaps and Separation. In *CP '13*. Springer, 282–298.
[20] Niklas Een, Alan Mishchenko, and Robert Brayton. 2011. Efficient Implementation of Property Directed Reachability. In *FMCAD '11* (Austin, Texas). IEEE, 125–134.
[21] Zafer Esen and Philipp Rümmer. 2022. Tricera: Verifying C Programs Using the Theory of Heaps. In *FMCAD '22*. 380–391.
[22] Jean-Christophe Filliâtre and Andrei Paskevich. 2013. Why3 - Where Programs Meet Provers. In *ESOP '13 (LNCS, Vol. 7792)*. Springer, 125–128.
[23] Cormac Flanagan and K. Rustan M. Leino. 2001. Houdini, an Annotation Assistant for ESC/Java. In *FME '01* (Berlin, Germany, 12–16) *(LNCS, Vol. 2021)*. Springer, 500–517.
[24] Timothy S. Freeman and Frank Pfenning. 1991. Refinement types for ML. In *PLDI '91* (Toronto, Ontario, Canada). ACM, 268–277.
[25] Lennard Gäher, Michael Sammler, Ralf Jung, Robbert Krebbers, and Derek Dreyer. 2024. RefinedRust: A Type System for High-Assurance Verification of Rust Programs. *Proceedings of the ACM on Programming Languages* 8, PLDI (2024), 1115–1139.
[26] Colin S. Gordon. 2020. Lifting Sequential Effects to Control Operators. In *ECOOP '20 (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 166)*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 23:1–23:30.
[27] Colin S. Gordon, Michael D. Ernst, and Dan Grossman. 2013. Rely-guarantee references for refinement types over aliased mutable data. In *PLDI '13* (Seattle, Washington, USA) *(PLDI '13)*. ACM, 73–84.
[28] Arie Gurfinkel, Temesghen Kahsai, Anvesh Komuravelli, and Jorge A. Navas. 2015. The SeaHorn Verification Framework. In *CAV '15*. Springer, 343–361.
[29] Son Ho, Aymeric Fromherz, and Jonathan Protzenko. 2024. Sound Borrow-Checking for Rust via Symbolic Semantics. *Proceedings of the ACM on Programming Languages* 8, ICFP, Article 251 (Aug. 2024), 29 pages.
[30] Son Ho and Jonathan Protzenko. 2022. Aeneas: Rust verification by functional translation. *Proceedings of the ACM on Programming Languages* 6, ICFP (2022), 711–741.

[31] Guilhem Jaber and Colin Riba. 2021. Temporal Refinements for Guarded Recursive Types. In *ESOP '21*. Springer, 548–578.
[32] Ralf Jung, Hoang-Hai Dang, Jeehoon Kang, and Derek Dreyer. 2020. Stacked borrows: an aliasing model for Rust. *Proceedings of the ACM on Programming Languages* 4, POPL (2020), 41:1–41:32.
[33] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2018. RustBelt: securing the foundations of the Rust programming language. *Proceedings of the ACM on Programming Languages* 2, POPL (2018), 66:1–66:34.
[34] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. 2018. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming* 28 (2018), e20.
[35] Ralf Jung, Rodolphe Lepigre, Gaurav Parthasarathy, Marianna Rapoport, Amin Timany, Derek Dreyer, and Bart Jacobs. 2019. The future is ours: prophecy variables in separation logic. *Proceedings of the ACM on Programming Languages* 4, POPL, Article 45 (Dec. 2019), 32 pages.
[36] Temesghen Kahsai, Philipp Rümmer, Huascar Sanchez, and Martin Schäf. 2016. JayHorn: A Framework for Verifying Java programs. In *CAV '16*, Vol. 9779. Springer, 352–358.
[37] Fuga Kawamata, Hiroshi Unno, Taro Sekiyama, and Tachio Terauchi. 2024. Answer Refinement Modification: Refinement Type System for Algebraic Effects and Handlers. *Proceedings of the ACM on Programming Languages* 8, POPL, Article 5 (Jan. 2024), 33 pages.
[38] Naoki Kobayashi, Ryosuke Sato, and Hiroshi Unno. 2011. Predicate abstraction and CEGAR for higher-order model checking. In *PLDI '11* (San Jose, California, USA). ACM, 222–233.
[39] Anvesh Komuravelli, Arie Gurfinkel, and Sagar Chaki. 2014. SMT-Based Model Checking for Recursive Programs. In *CAV '14 (LNCS, Vol. 8559)*. Springer, 17–34.
[40] Hari Govind Vediramana Krishnan, Yuting Chen, Sharon Shoham, and Arie Gurfinkel. 2020. Global Guidance for Local Generalization in Model Checking. In *CAV '20 (LNCS, Vol. 12225)*. Springer, 101–125.
[41] Satoshi Kura and Hiroshi Unno. 2024. Automated Verification of Higher-Order Probabilistic Programs via a Dependent Refinement Type System. *Proceedings of the ACM on Programming Languages* 8, ICFP, Article 269 (Aug. 2024), 30 pages.
[42] Andrea Lattuada, Travis Hance, Chanhee Cho, Matthias Brun, Isitha Subasinghe, Yi Zhou, Jon Howell, Bryan Parno, and Chris Hawblitzel. 2023. Verus: Verifying Rust Programs using Linear Ghost Types. *Proceedings of the ACM on Programming Languages* 7, OOPSLA1 (2023), 286–315.
[43] Nico Lehmann, Adam T. Geller, Niki Vazou, and Ranjit Jhala. 2023. Flux: Liquid Types for Rust. *Proceedings of the ACM on Programming Languages* 7, PLDI (2023), 1533–1557.
[44] Marcus Lindner, Jorge Aparicius, and Per Lindgren. 2018. No Panic! Verification of Rust Programs by Symbolic Execution. In *2018 IEEE 16th International Conference on Industrial Informatics (INDIN)*. 108–114.
[45] Makai Mann, Ahmed Irfan, Alberto Griggio, Oded Padon, and Clark Barrett. 2022. Counterexample-Guided Prophecy for Model Checking Modulo the Theory of Arrays. *Logical Methods in Computer Science* Volume 18, Issue 3 (Aug. 2022).
[46] Yusuke Matsushita, Xavier Denis, Jacques-Henri Jourdan, and Derek Dreyer. 2022. RustHornBelt: a semantic foundation for functional verification of Rust programs with unsafe code. In *PLDI '22*. ACM, 841–856.
[47] Yusuke Matsushita, Takeshi Tsukada, and Naoki Kobayashi. 2021. RustHorn: CHC-based Verification for Rust Programs. *ACM Transactions on Programming Languages and Systems* 43, 4 (2021), 15:1–15:54.
[48] Peter Müller, Malte Schwerhoff, and Alexander J. Summers. 2016. Viper: A Verification Infrastructure for Permission-Based Reasoning. In *VMCAI '16* (St. Petersburg, FL, USA) *(LNCS, Vol. 9583)*. Springer, 41–62.
[49] Takashi Nakayama, Yusuke Matsushita, Ken Sakayori, Ryosuke Sato, and Naoki Kobayashi. 2024. Borrowable Fractional Ownership Types for Verification. In *VMCAI '24*. Springer, 224–246.
[50] Yoji Nanjo, Hiroshi Unno, Eric Koskinen, and Tachio Terauchi. 2018. A Fixpoint Logic and Dependent Effects for Temporal Property Verification. In *LICS '18*. ACM, 759–768.
[51] Hiromi Ogawa, Taro Sekiyama, and Hiroshi Unno. 2025. *Thrust: A Prophecy-based Refinement Type System for Rust - Artifact*. doi:10.5281/zenodo.15017946
[52] Patrick Rondon, Ming Kawaguchi, and Ranjit Jhala. 2008. Liquid Types. In *PLDI '08*. ACM, 159–169.
[53] Patrick Maxim Rondon, Ming Kawaguchi, and Ranjit Jhala. 2010. Low-level liquid types. In *POPL '10* (Madrid, Spain). ACM, 131–144.
[54] Michael Sammler, Rodolphe Lepigre, Robbert Krebbers, Kayvan Memarian, Derek Dreyer, and Deepak Garg. 2021. RefinedC: Automating the Foundational Verification of C Code with Refined Ownership Types. In *PLDI '21* (Virtual, Canada) *(PLDI 2021)*. ACM, 158–174.
[55] Taro Sekiyama and Hiroshi Unno. 2023. Temporal Verification with Answer-Effect Modification: Dependent Temporal Type-and-Effect System with Delimited Continuations. *Proceedings of the ACM on Programming Languages* 7, POPL, Article 71 (Jan. 2023), 32 pages.

[56] Frederick Smith, David Walker, and J. Gregory Morrisett. 2000. Alias Types. In *ESOP '00* (Berlin, Germany, March 25 – April 2) *(LNCS, Vol. 1782)*. Springer, 366–381.
[57] Nikhil Swamy, Joel Weinberger, Cole Schlesinger, Juan Chen, and Benjamin Livshits. 2013. Verifying higher-order programs with the dijkstra monad. In *PLDI '13* (Seattle, Washington, USA) *(PLDI '13)*. ACM, 387–398.
[58] The Rust Compiler Team. 2022. *The Polonius Book*. https://rust-lang.github.io/polonius/.
[59] John Toman, Stuart Pernsteiner, and Emina Torlak. 2015. Crust: A Bounded Verifier for Rust (N). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 75–80.
[60] John Toman, Ren Siqi, Kohei Suenaga, Atsushi Igarashi, and Naoki Kobayashi. 2020. ConSORT: Context- and Flow-Sensitive Ownership Refinement Types for Imperative Programs. In *ESOP '20 (LNCS, Vol. 12075)*. Springer, 684–714.
[61] Linus Torvalds. 2022. *Merge Commit for initial Rust support in the Linux kernel*. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=8aebac82933ff1a7c8eede18cab11e1115e2062b.
[62] Takeshi Tsukada and Hiroshi Unno. 2024. Inductive Approach to Spacer. *Proceedings of the ACM on Programming Languages* 8, PLDI, Article 227 (June 2024), 24 pages.
[63] Sebastian Andreas Ullrich. 2016. Simple Verification of Rust Programs via Functional Purification.
[64] Hiroshi Unno and Naoki Kobayashi. 2009. Dependent Type Inference with Interpolants. In *PPDP '09* (Coimbra, Portugal). ACM, 277–288.
[65] Hiroshi Unno, Yuki Satake, and Tachio Terauchi. 2017. Relatively Complete Refinement Type System for Verification of Higher-order Non-deterministic Programs. *Proceedings of the ACM on Programming Languages* 2, POPL, Article 12 (Dec. 2017), 29 pages.
[66] Hiroshi Unno, Tachio Terauchi, and Naoki Kobayashi. 2013. Automating Relatively Complete Verification of Higher-order Functional Programs. In *POPL '13* (Rome, Italy). ACM, 75–86.
[67] Hiroshi Unno, Tachio Terauchi, and Eric Koskinen. 2021. Constraint-Based Relational Verification. In *CAV '21*. Springer, 742–766.
[68] Alexa VanHattum, Daniel Schwartz-Narbonne, Nathan Chong, and Adrian Sampson. 2022. Verifying dynamic trait objects in Rust *(ICSE-SEIP '22)*. ACM, 321–330.
[69] Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon L. Peyton Jones. 2014. Refinement Types for Haskell. In *ICFP '14* (Gothenburg, Sweden). ACM, 269–282.
[70] Fabian Wolff, Aurel Bílý, Christoph Matheja, Peter Müller, and Alexander J. Summers. 2021. Modular specification and verification of closures in Rust. *Proceedings of the ACM on Programming Languages* 5, OOPSLA, Article 145 (Oct. 2021), 29 pages.
[71] Andrew K. Wright and Matthias Felleisen. 1994. A Syntactic Approach to Type Soundness. *Inf. Comput.* 115, 1 (1994), 38–94. doi:10.1006/inco.1994.1093
[72] Hongwei Xi. 2001. Dependent Types for Program Termination Verification. In *LICS '01*. IEEE, 231–242.
[73] Hongwei Xi and Frank Pfenning. 1998. Eliminating array bound checking through dependent types. In *PLDI '98* (Montreal, Quebec, Canada). ACM, 249–257.
[74] Dengping Zhu and Hongwei Xi. 2005. Safe Programming with Pointers through Stateful Views. In *PADL '05* (Long Beach, CA, USA, 10–11) *(LNCS, Vol. 3350)*. Springer, 83–97.

Fig. 11. Example of drop points analysis in Thrust

```
let mut _0: ();
let mut _2: bool;
let mut _3: i64;

bb0: { // { _0, _1 }
  _3 = (*_1);
  // { _0, _1, _3 }
  _2 = Ge(move _3, const 0_i64);
  // { _0, _1, _2 }
  switchInt(move _2)
  // { _0, _1 }
    -> [0: bb2,
        otherwise: bb1];
}

bb1: { // { _0, _1 }
  (*_1) = const 0_i64;
  // { _0 }
  goto -> bb2;
  // { _0 }
}

bb2: { // { _0 }
  return;
  // {}
}
```

```
let mut _0: ();
let mut _2: bool;
let mut _3: i64;

bb0: {
  _3 = (*_1);

  _2 = Ge(move _3, const 0_i64);
  // drop(_3);
  switchInt(move _2)
  // drop(_2);
    -> [0: bb2, // drop(_1);
        otherwise: bb1];
}

bb1: {
  (*_1) = const 0_i64;
  // drop(_1);
  goto -> bb2;

}

bb2: {
  return;
  // drop(_0);
}
```

A Implementation Details: Differences Between λ_{COR} and MIR

We had to handle many differences between MIR and λ_{COR} to implement Thrust on MIR. First of all, MIR is a control-flow graph (CFG)-based representation; each function body consists of one or more basic blocks, and each basic block consists of zero or more statements and one terminator. The terminator is a control-flow instruction, such as a conditional branch or a jump, which determines how the control flows between basic blocks. By contrast, λ_{COR} is based on expressions that rely on conditional expressions and recursive functions to express control flows. To fill this gap, Thrust interprets MIR as a functional program in a manner discussed by Appel [2]. Other differences and how Thrust deals with them are explained in the following.

Finding drop points. As explained earlier, drops are explicit in λ_{COR}, whereas they are not in Rust. The optimized MIR does not contain explicit drops either. Thus, Thrust needs to determine where to drop variables before typing. To achieve this, Thrust utilizes live-variable analysis and inserts drops at points where a variable transitions from live to dead. Drop points can occur either immediately after a statement or on the edges of the CFG. Figure 11 illustrates how drop points are identified in a MIR body. The left side shows a source MIR with the results of live-variable analysis written as comments, while the right side depicts the MIR with drops inserted by Thrust. Thrust does not actually modify the MIR to insert drops. Instead, it performs the analysis before typing to

identify drop points and incorporates appropriate assumptions into typing, as if the drops were performed according to the identified points.

Fig. 12. Example of automatic reborrows in THRUST

```
1
2  (*_1) = const 0_i64;
3
4  _2 = set(_1) -> [
5    return: bb1,
6    unwind continue
7  ];
```

```
1  _3 = &mut _1;
2  (*_3) = const 0_i64;
3  _4 = &mut _1;
4  _2 = set(move _4) -> [
5    return: bb1,
6    unwind continue
7  ];
```

Inserting reborrows. λ_{COR} only allows a single assignment to a mutable reference; therefore, reborrows are required to perform multiple mutations. However, in RUST and MIR, multiple mutations on a single mutable reference are allowed. To bridge this gap, THRUST automatically inserts reborrows when a place appears on the left-hand side of an assignment. When an assignment is performed, THRUST inserts a reborrow just before the assignment and replaces the left-hand side place with the newly created reference. This approach allows the original reference to remain alive during typing while preserving the effect of the assignment. THRUST also inserts reborrows when mutable references are copied as operands. This behavior might seem unusual, as mutable references are generally not expected to be copied. However, these copies originate from a RUST mechanism known as *reborrow coercions*. In RUST, values of type `&'a mut` can be coerced [14] to `&'b mut` when `'a` outlives `'b`. This coercion effectively results in a reborrow, as indicated by the change in lifetimes. Such implicit reborrows are present in MIR when it is initially constructed, but they are replaced with copies during optimization. Since THRUST processes MIR after optimization, we needed to reverse this optimization by treating copied mutable references as reborrows. Figure 12 provides an example of these reborrows automatically inserted by THRUST. The left program shows the source MIR, while the right one depicts the MIR with reborrows inserted by THRUST.

Mutable local variables. In λ_{COR}, we may only borrow pointers or tuples. On the other hand, borrowing ordinary local variables is allowed in RUST and MIR. To bridge this gap, THRUST treats every local variable marked `mut` as owned pointers during typing. When these variables appear in right-hand side expressions, THRUST interprets their values as if they were dereferenced to access the inner value.

Enums. Although λ_{COR} includes tuples, it does not include general algebraic data types (ADTs). On the other hand, the implementation of THRUST does support ADTs. THRUST primarily encodes ADTs directly into the ADT theory of the backend solver. However, to support mutation of ADT elements, a method to handle disjoint sum types (RUST's "enum") within the place context is required. To address this, THRUST adds new entries for enums to the place context, as shown below.[11]

$$\Delta ::= \cdots \mid x : x_d @ (V_1(x_1) \mid V_2(x_2) \mid \cdots \mid V_n(x_n))$$

Here, $V_1, \ldots, V_n$ are the names of each variant. x_d is an implicit integer variable for the discriminant of the enum, and x_1 through x_n are implicit variables for the contents of each variant. By expressing

[11]For simplicity, we here assume that each variant has a single field.

it in this way, borrowing from the contents of each variant can be handled in the same way as with tuples. To bind a variable of enum type in this place context, we need to express the refinement of that variable using these implicit variables. For example, consider an enum D and assume it is bound with the type $\{x : D \mid \varphi\}$. After binding the implicit variables $x_1, \ldots, x_n$, which correspond to the variants $V_1, \ldots, V_n$ of D, and x_d, which corresponds to the discriminant, we add the following assumption to the environment:

$$\exists x.\ m_D(x_1, \ldots, x_n, x) \land \text{discriminant}_D(x) = x_d \land \varphi$$

where $\text{discriminant}_D(x)$ is a function that returns the discriminant of x, and m_D is a predicate that associates x with $x_1, \ldots, x_n$, defined as follows for each enum:

$$m_D(x_1, \ldots, x_n, x) = (V_1(x_1) = x \lor V_2(x_2) = x \lor \cdots \lor V_n(x_n) = x)$$

When using a variable of enum type bound in this way in the place context, we must also represent the refinement of type D using these implicit variables. Given the following environment:

$$\Gamma = x_d : \text{int}, x_1 : U_1, \ldots, x_n : U_n$$
$$\Delta = \cdots \mid x : x_d@(V_1(x_1) \mid V_2(x_2) \mid \cdots \mid V_n(x_n))$$

The place typing of x is as follows.

$$\Gamma \mid \Delta \vdash x : D \ \triangleright \exists x.\ x \mid m_D(x_1, \ldots, x_n, x) \land \text{discriminant}_D(x) = x_d$$

To handle this, place typing is extended to take existentially quantified variables and additional logical formulas.

Recursive ADTs. Thrust also supports recursive ADTs, such as lists. However, when attempting to bind recursive ADTs to a place context as described above, they can expand indefinitely. Currently, Thrust addresses this issue by predefining a depth limit for recursive expansions. If an attempt is made to borrow a part of the place context that has not been expanded, constraint generation fails, limiting the range of MIR that can be handled. However, this is not a significant problem, as one can simply bind the desired depth to a variable if a deeper borrow is needed. Additionally, we believe that the depth limit can be determined through static analysis of where borrowing occurs for each variable, and we plan to implement this approach in the future.

B Full Experiment Results on the RustHorn Benchmark Set

The experiment results on the RustHorn benchmark set are shown in Table 3. The column "Result" indicates the verification outcome provided by the tool, showing whether the tool identifies the program as safe or unsafe. Some results indicate an abort or a timeout. The column "Time(s)" represents the elapsed verification time in seconds.

Table 3. Experiment results on the RustHorn benchmark set

	Thrust		RustHorn	
	Result	Time(s)	Result	Time(s)
simple-1-01_unsat	safe	0.06	safe	1.3
simple-2-04_recursive_unsat	safe	1.99	safe	0.35
simple-3-05_recursive_sat	unsafe†	0.06	unsafe	0.07
simple-4-06_loop_unsat	timeout	N/A	timeout	N/A
simple-5-hhk2008	safe	0.06	timeout	N/A
simple-6-unique_scalar	unsafe†	0.05	unsafe	0.06
bmc-1-test-bmc-1-safe	safe	0.13	safe	0.15
bmc-1-test-bmc-1-unsafe	unsafe†	0.13	unsafe	0.14
bmc-2-test-bmc-2-safe	safe	0.06	safe	0.07
bmc-2-test-bmc-2-unsafe	unsafe†	0.07	unsafe	0.07
bmc-3-test-bmc-3-safe	safe	0.09	safe	0.09
bmc-3-test-bmc-3-unsafe	unsafe†	0.08	unsafe	0.09
bmc-4-test-bmc-diamond-1-safe	safe	0.11	safe	0.09
bmc-4-test-bmc-diamond-1-unsafe	unsafe†	0.1	unsafe	0.09
bmc-5-test-bmc-diamond-2-safe	safe	0.1	safe	0.1
bmc-5-test-bmc-diamond-2-unsafe	unsafe†	0.1	unsafe	0.09
prusti-1-pass-rosetta-Ackermann_function-base	safe	0.08	safe	0.06
prusti-2-pass-rosetta-Ackermann_function-same	timeout	N/A	timeout	N/A
prusti-3-pass-paper_examples-points-compress	safe	0.07	safe	0.1
prusti-4-pass-paper_examples-borrows_align	safe	0.07	safe	0.05
prusti-5-pass-demos-account	safe	0.08	safe	0.04
prusti-6-fail-demos-account_error_1	unsafe†	0.08	unsafe	0.05
prusti-7-pass-mut_borrows-restore	safe	0.09	safe	0.05
inc-max-1-base-safe	safe	0.06	safe	0.05
inc-max-1-base-unsafe	unsafe†	0.06	unsafe	0.05
inc-max-2-base3-safe	safe	0.1	safe	0.07
inc-max-2-base3-unsafe	unsafe†	0.1	unsafe	0.07
inc-max-3-repeat-safe	safe	0.26	safe	0.1
inc-max-3-repeat-unsafe	unsafe†	0.08	unsafe	0.05
inc-max-4-repeat3-safe	safe	0.51	safe	0.13
inc-max-4-repeat3-unsafe	unsafe†	0.09	unsafe	0.06
swap-dec-1-base-safe	timeout	N/A	timeout	N/A
swap-dec-1-base-unsafe	timeout	N/A	timeout	N/A
swap-dec-2-base3-safe	safe	3.41	timeout	N/A
swap-dec-2-base3-unsafe	unsafe†	16.03	timeout	N/A
swap-dec-3-exact-safe	safe	0.32	timeout	N/A
swap-dec-3-exact-unsafe	unsafe†	0.13	unsafe	0.06
swap-dec-4-exact3-safe	timeout	N/A	timeout	N/A
swap-dec-4-exact3-unsafe	abort	N/A	unsafe	0.07
swap2-dec-1-base-safe	safe	1.3	timeout	N/A
swap2-dec-1-base-unsafe	unsafe†	2.79	timeout	N/A
swap2-dec-2-base3-safe	timeout	N/A	safe	0.6
swap2-dec-2-base3-unsafe	timeout	N/A	unsafe	2.44
swap2-dec-3-exact-safe	timeout	N/A	timeout	N/A
swap2-dec-3-exact-unsafe	unsafe†	0.58	unsafe	0.11
swap2-dec-4-exact3-safe	timeout	N/A	timeout	N/A
swap2-dec-4-exact3-unsafe	timeout	N/A	unsafe	0.07
just-rec-1-base-safe	safe	0.07	safe	0.05
just-rec-1-base-unsafe	unsafe†	0.1	unsafe	0.05
linger-dec-1-basic-safe	safe	0.1	safe	0.05
linger-dec-1-basic-unsafe	unsafe†	0.11	unsafe	0.05
linger-dec-2-basic3-safe	safe	0.34	safe	0.06
linger-dec-2-basic3-unsafe	unsafe†	0.34	unsafe	0.05
linger-dec-3-exact-safe	safe	0.12	safe	0.06
linger-dec-3-exact-unsafe	unsafe†	0.1	unsafe	0.05
linger-dec-4-exact3-safe	safe	0.27	safe	0.07
linger-dec-4-exact3-unsafe	unsafe†	0.24	unsafe	0.06
lists-1-append-safe	abort	N/A	abort	N/A
lists-1-append-unsafe	unsafe†	8.3	unsafe	0.06
lists-2-inc-all-safe	abort	N/A	timeout	N/A
lists-2-inc-all-unsafe	unsafe†	2.77	unsafe	0.06
lists-3-inc-some-safe	abort	N/A	timeout	N/A
lists-3-inc-some-unsafe	unsafe†	9.33	unsafe	0.07
lists-4-inc-some2-safe	abort	N/A	abort	N/A
lists-4-inc-some2-unsafe	unsafe†	12.97	unsafe	0.1
trees-1-append-safe	timeout	N/A	timeout	N/A
trees-1-append-unsafe	unsafe†	175.94	unsafe	0.06
trees-2-inc-all-safe	abort	N/A	abort	N/A
trees-2-inc-all-unsafe	unsafe†	13.74	unsafe	0.06
trees-3-inc-some-safe	timeout	N/A	timeout	N/A
trees-3-inc-some-unsafe	timeout	N/A	unsafe	0.08
trees-4-inc-some2-safe	timeout	N/A	abort	N/A
trees-4-inc-some2-unsafe	timeout	N/A	unsafe	0.11

† We manually confirmed that the program is indeed unsafe when Thrust concludes that the program is unsafe, since we have not formally proved the completeness of Thrust, although it is conjectured to be complete for first-order programs for the same reason that RustHorn is.