Ravencheck: Effectively-Propositional Reasoning for Rust

Kunha Kim
University of Colorado Boulder
Boulder, CO, USA
kunha.kim@colorado.edu

Nicholas V. Lewchenko
University of Colorado Boulder
Boulder, CO, USA
nicholas.lewchenko@colorado.edu

Bor-Yuh Evan Chang
University of Colorado Boulder
Boulder, CO, USA and Amazon, USA
evan.chang@colorado.edu

Gowtham Kaki
University of Colorado Boulder
Boulder, CO, USA
gowtham.kaki@colorado.edu

Abstract

SMT-based automated verification is a promising approach for ensuring software correctness, but it often suffers from unpredictability and proof instability due to the undecidability of the underlying logic. While tools like Liquid Haskell or F* provide automation, they either restrict logical expressivity to ensure decidability or sacrifice decidability and predictability for richer logic. In this paper, we present Ravencheck, a verification tool for Rust programs. Ravencheck occupies a unique design space: it supports specifications within an expressive fragment of Higher-Order Logic (HOL) while guaranteeing decidable verification outcomes. To achieve this, Ravencheck automatically encodes high-level Rust programs into the Extended Effectively Propositional (EEPR) fragment of first-order logic with relational abstraction.

Ravencheck builds on prior EEPR-based modeling languages (e.g., Ivy) by integrating with the standard Rust toolchain. This integration supports a one-language workflow in which verification conditions are expressed using the same purely functional Rust syntax as the verified executable code, rather than introducing a separate modeling language.

1 Introduction

Logical deduction aided by Satisfiability-Modulo-Theory (SMT) solvers is the cornerstone of modern automated software verification. Language-integrated verification frameworks, such as F* [22], Dafny [14], and Verus [13], have successfully integrated solver-aided reasoning into software development process, enabling formal verification to be applied at scale to industry-grade software [9, 21]. Despite these advances, the adoption of such tools in mainstream software development remains limited. Among the various challenges, a key obstacle is the unpredictability of automated reasoning. Because the logics used in program verification—typically first-order logic with quantifiers and background theories—are undecidable, SMT solvers must rely on heuristic strategies that can be brittle. As a result, verification efforts often suffer from proof instability [26], where semantically irrelevant modifications to code or specifications, such as refactoring the function definitions, can cause previously successful proofs to fail or time out.

```
use std::collections::HashSet;
#[declare]
fn insert(e: u32, mut s: HashSet<u32>)
        -> HashSet<u32> {
  s.insert(e);
  s
}
// ¬contains(s1, 4)
let s2 = insert(s1, 4);
// ¬contains(s1, 4) ∧ contains(s2, 4)
```

Figure 1. Rust's ownership system allows `insert` to be treated as a pure function despite delegating to `HashSet::insert`.

To mitigate this problem, verification frameworks often force a trade-off between the expressive power (of programs and specifications) and automation (of proofs). For example, Liquid Haskell [25] restricts type refinements to a quantifier-free fragment of first-order logic, limiting the expressivity of the specification language but ensuring the decidability of type checking. On the other hand, F* [22] admits unrestricted quantification and higher-order functions, but loses decidability of type checking. Quantifiers are indispensable for expressing non-trivial correctness properties, such as heap reachability, yet decidability is equally essential to ensure that automated reasoning about these properties terminates predictably. Removing the predictability of automation forces programmers to either manually guide the solver or contend with "butterfly effects" in proof maintenance.

In this paper, we present Ravencheck – a language-integrated verification framework for Rust that helps programmers bridge the gap between expressive high-level abstractions and restrictive decidable logics. Ravencheck allows Rust code composed of higher-order functions and algebraic data types to be verified against quantified specifications — also written in Rust – while guaranteeing that the verification conditions remain within the Extended Effectively Propositional (EEPR) fragment of first-order logic [12].

A notable aspect of Ravencheck is that it leverages Rust's

ownership type system to soundly interface with code that mutates the heap. Fig. 1 shows an example. Here we use Rust's standard library `HashSet` to implement a functional Set data type. The Set `insert` function calls `HashSet::insert` to in-place update the set and return a reference. Consequently, `s1` and `s2` are aliases to the same set. However, Rust's ownership type system prevents `s1` from being used in the continuation of the call to `insert`, letting us soundly pretend that `s1` and `s2` are distinct sets and `insert` is a pure function. This style of imbuing a functional interface to mutable implementations allows programmers to leverage automated reasoning without sacrificing performance.

Architecture. Figure 2 illustrates the high-level architecture of Ravencheck. The tool takes as its input Rust source files composed of function definitions, checkable specifications, and axioms, all expressed in the Rust syntax. The tool extracts a *verification projection* of the source code consisting of functions that need to be verified, their specifications, and axioms. The function definitions that are not marked for verification are elided from the verification projection and treated as uninterpreted functions. They are however included in the *execution projection*, which includes the executable source code except specifications and axioms. The verification projection is partially evaluated by a Call-By-Push-Value (CBPV) abstract machine [15], transforming it first into an A-normal intermediate representation and finally into relationally abstracted first-order encoding. The abstracted formulas are checked by off-the-shelf SMT solvers(e.g., Z3 [5], CVC5 [2]) to produce a validity result or a counterexample.

1.1 Related Work

Ravencheck is inspired by Ivy [19]: a widely-known modeling and verification tool for distributed systems that produces verification conditions in the Extended EPR fragment. While Ravencheck shares Ivy's foundational goal of predictable automation via the Extended EPR fragment, it fundamentally differs in its user-facing design. Ivy users program in a unique specification language, relying on a C++ transpilation mechanism to generate executable code. In contrast, Ravencheck allows developers to write and verify code directly in the familiar Rust language. By unifying executable code and specifications within Rust, Ravencheck eliminates the need for a verification-specific modeling language and provides a simple, single-language verification workflow.

Existing verification tools that integrate with Rust, such as Verus [13], focus on advanced language features, such as `unsafe` blocks, at the expense of direct, decidable proof automation. Similarly, tools based on separation logic, such as Prusti [1], prioritize expressive memory modeling over predictable automation, leaving the verification process susceptible to solver heuristics and proof instability. Creusot [7] translates Rust into the WhyML language to model mutable borrows with prophecies, and requires developers to provide manual proofs in the Why3 IDE to verify complex properties. Ravencheck is distinguished from these tools by its emphasis on predictable automation and a single-language workflow. By implementing an automatic EPR encoding for a commonly-used subset of Rust, Ravencheck ensures decidability, and any necessary manual effort—such as quantifier instantiation—is resolved seamlessly using familiar Rust constructs.

2 An Overview of Ravencheck

Unlike many verification tools that require developers to learn a separate modeling language or deal with the unpredictability of general SMT solving, Ravencheck integrates directly into the standard Rust ecosystem. It leverages Rust's procedural macro system to allow developers to write specifications and verification conditions alongside their executable code. This approach ensures that verified code can be easily integrated with ordinary Rust projects, while the underlying analysis guarantees that all verification queries remain within the decidable Extended EPR fragment. Specifically, Ravencheck's type checker enforces this guarantee by statically analyzing quantifier alternations for sort cycles and reporting compile-time errors to keep specifications within the decidable fragment. In this section, we describe Ravencheck's Rust interface and verification workflow with help of illustrative examples.

2.1 Basic Interface

We illustrate Ravencheck's interface using a heap manipulation example (Fig. 3), which defines a verifiable swap operation on a vector-backed heap.

Module Setup. The primary unit of verification in Ravencheck is a Rust module annotated with the `#[ravencheck::module]` attribute. The annotation triggers the generation of a test module that invokes the verification engine when the standard `cargo test` command is run. Verification is thus integrated into Rust's existing build-and-test workflow.

Declarations and Axioms. Ravencheck allows users to declare uninterpreted types and functions using the `#[declare]` attribute. In Fig. 3, the type `Heap` is treated as an abstract sort by the solver, decoupling the verification logic from the actual `Vec<Val>` implementation. Similarly, the primitive operations `get` and `set` shown in the listing are declared as uninterpreted functions. Such functions are required to expose a purely functional interface (e.g., `set` returns a new `Heap`), though their implementations may use in-place mutation. The behavior of declared (uninterpreted) functions can be axiomatized using the `#[assume]` attribute. In Fig. 3, `ax_heap` relates the behavior of `get` and `set` functions using standard array axioms. For this running example, we will assume that all `Heap` vectors are long enough for all `Ptr`

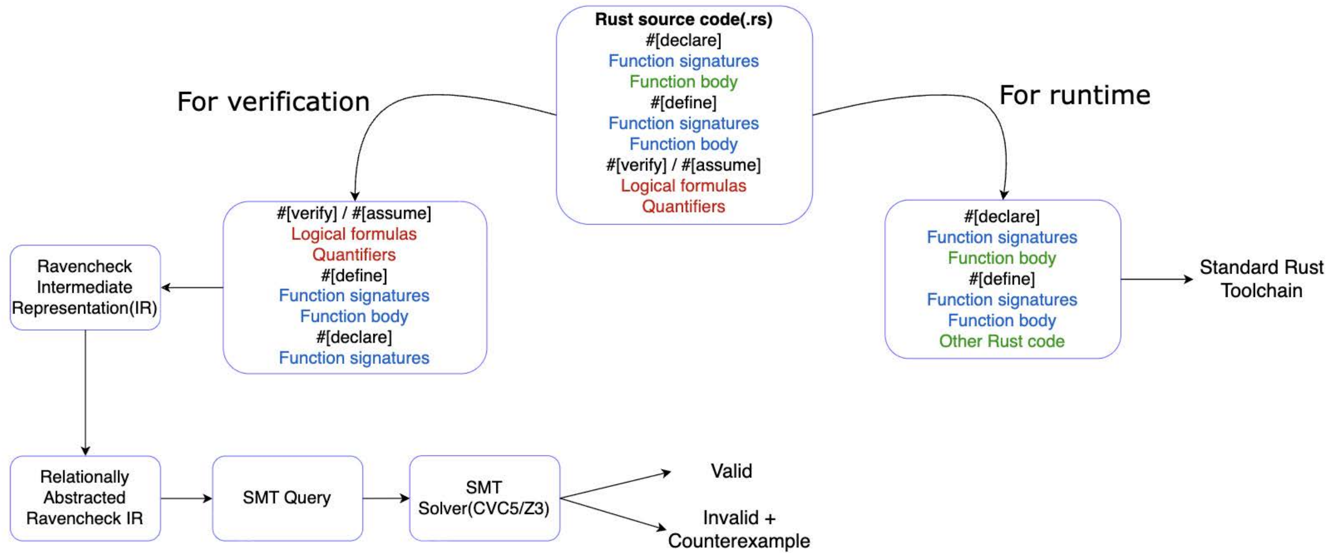

Figure 2. High-level architecture of Ravencheck. The tool splits a Rust module into a verification projection (left) and an executable projection (right). The red part is lowered into a CBPV-based IR for SMT solving, while the green part is compiled via the standard Rust toolchain. The blue part remains shared across both streams.

```
#[declare]
type Heap = Vec<Val>;
#[declare]
#[total]
pub fn get(h: &Heap, p: Ptr) -> Val {
    h[p]
}
#[declare]
pub fn set(mut h: Heap, p: Ptr, val: Val) -> Heap {
    h[p]=val;
    h
}
#[assume]
fn ax_heap(p1: Ptr, h: Heap, p: Ptr, v: Val) -> bool
    if p1 == p {
        get(set(h, p, val), p1) == val
    } else {
        get(set(h, p, val), p1) == get(h, p1)
    }
}
```

Figure 3. Declarations and axioms of `get` and `set`. Note that Rust's reference operator (&) can be omitted within axioms.

indices to be within-bounds. Note the `#[total]` annotation on `get`: while both `get` and `set` are total functions (under our example's assumptions), decidable verification requires the solver to assume that `set` may be partial. We will discuss this further in Sec. 2.3.

The arguments of `ax_heap` are implicitly universally quantified, but universal and existential quantifiers can also be explicitly invoked. In Fig. 4, we use nested quantifiers in an axiom to assume that a minimal (zero) pointer exists, for a hypothetical `less_or_eq` relation. The colored backgrounds

```
#[assume]
pub fn minimal_ptr() -> bool {
    exists(|p0: Ptr| {
        forall(|p1: Ptr| less_or_eq(p0,p1))
    )}
}
```

Figure 4. Explicit quantifiers.

```
#[define]
pub fn swap(h: Heap, a: Ptr, b: Ptr) -> Heap {
    let next_a = get(&h, a);
    let next_b = get(&h, b);
    let h1 = set(h, a, next_b);
    set(h1, b, next_a)
}
```

Figure 5. Swap is defined in terms of `get` and `set`.

in our code examples indicate which parts of the code are used for verification (red and blue) and for runtime execution (green and blue), as established in Fig. 2. The lines in red, which cover all quantifiers, are erased in the execution projection before being passed on to the Rust compiler. The lines in white indicates Rust macros that are interpreted by Ravencheck and subsequently erased.

Definitions. The `#[define]` attribute allows users to define *interpreted* functions, with bodies that are visible to the verification process. This is useful for defining helper functions or shorthands, such as `swap` in Fig. 5. When `swap` appears in a verification condition, its definition is inlined, so that its behavior is known in terms of the `get` and `set`

```
#[verify]
fn swap_commute() -> bool {
    forall(|h: Heap, a: Ptr, b: Ptr, c: Ptr, d: Ptr, k: Ptr|{
        let disjoint = a != c && a != d && b != c && b != d;
            // If (a,b) are disjoint from (c,d) ...
        implies(disjoint, {
            // ... then swap on (a,b) commutes with swap on (c,d).
            let lhs = swap(swap(h, a, b), c, d);
            let rhs = swap(swap(h, c, d), a, b);
            get(lhs, k) == get(rhs, k)
        })
    })
}
```

Figure 6. Verification condition on swap.

```
#[declare]
pub fn filter(f: fn(Elem) -> bool, s: Set) -> Set {
    ...
}
#[assume(filter(f, s1) => s2)]
fn filter_def() -> bool {
    forall(|e: u32| {
        member(e,s2) == (member(e,s1) && f(e))
    })
}
```

Figure 7. Axiomatization of higher-order function, using restricted higher-order quantification.

calls within. Because defined functions are processed by the verification engine, their bodies are restricted: they cannot contain mutation or method calls (that use the dot operator, e.g., `self.f()`). Recursion is allowed as a special case, which we will discuss in Sec. 2.2. Defined functions can be shared between verification conditions and executable code as long as they do not contain quantifiers.

Verification Conditions. The `#[verify]` attribute marks a function as a property to be proven, analagous to the `#[test]` attribute that Rust programmers already use to declare runtime tests. In Fig. 6, `swap_commute` asserts that the order of two `swap` operations does not matter if they operate on disjoint indices. To validate this, our example demonstrates the pattern of proving *observational equivalence* by asserting that `get(lhs, k) == get(rhs, k)` holds for any arbitrary index `k`.

Higher-Order Functions. Ravencheck code occupies a fragment of Higher Order Logic (HOL): higher-order functions can be defined, but higher-order quantification is restricted. Fig. 7 gives an example of how to axiomatize a higher-order set filter function.

Our `forall`/`exists` terms cannot quantify `filter`'s function input `f`, and so we universally quantify `f` at the top-level, in the `#[assume(filter(f, s1) => s2)]` attribute. Higher-order functions can be freely introduced using `#[define]`, since those do not need to be axiomatized, and they are

```
#[define]
enum Nat { Z, S(Box<Nat>) }
#[define]
#[recursive]
fn add(a: Nat, b: Nat) -> Nat {
    match a {
        Nat::Z => b,
        Nat::S(a_m) =>
            Nat::S(Box::new(add(*a_m, b)))
    }
}
```

Figure 8. The recursive definition of add.

```
#[annotate]
#[forall_inductive(a: Nat, b: Nat)]
fn factor_s_right() -> bool {
    Nat::S(add(a,b)) == add(a,Nat::S(b))
}
```

Figure 9. Annotation for add. Note the omission of the `Box::new` constructor.

erased by inlining before the verification condition is encoded to first-order SMT.

2.2 Enums and Recursive Functions

Unlike non-recursive function definitions, recursive definitions cannot be inlined as it would lead to infinite expansion. Recursive functions are therefore encoded as uninterpreted functions with (optional) annotations characterizing their behavior. Such annotations, if present, are verified against the function definition via induction. Fig. 8 illustrates with help of using the example of Peano arithmetic. The inductive definition of `Nat` is encoded as a Rust `enum` type. The addition operation is defined by the recursive function `add`, marked with the `#[recursive]` attribute. Fig. 9 shows a possible annotation for add that user might want to verify. The annotation effectively asserts that `S(add(a,b)) === add(a, S(b))` forall `Nat`s `a` and `b` introduced by the attribute `#[forall_inductive(a: Nat, b: Nat)]`.

The attribute also exposes the inductive structure of `a` and `b`, allowing induction to be performed. Since `add` is marked with `#[define]]`, its definition is unrolled when the function is applied to terms of the form `Z` and `S(. . . )`. The unrolling, together with the inductive hypothesis is sufficient for the solver to discharge the proof obligation generated by `#[annotate]`. Note that the `Nat` constructor `S` is applied directly to terms of type `Nat` in Fig. 9 whereas its `enum` type definition (Fig. 8) requires it to be applied to a `Box<Nat>`. Indeed, `factor_s_right` does not type-check if passed to the Rust compiler. However, as the color highlighting indicates, the propositions marked with `#[annotate]` are not passed to the compiler, so the type error never manifests. Ravencheck does not distinguish between the types `Box<T>`

```
#[assume]
fn add_commutative() -> bool {
    forall(|x: Nat, y: Nat|{
        add(x,y) == add(y,x)
    })
}
#[assume]
fn factor_s_left() -> bool {
    forall(|x: Nat, y: Nat|{
        add(Nat::S(x),y) == Nat::S(add(x,y))
    })
}
#[verify]
fn factor_s_right() -> bool {
    forall(|a: Nat, b: Nat| {
        let _ = add(Nat::S(b), a);
        let _ = Nat::S(add(b,a));
        add(a, Nat::S(b)) == Nat::S(add(a,b))
    })
}
```

Figure 10. Quantifier instantiation via let binding.

and `T`, and treats calls to `Box::new` and `clone` as identity operations, so the code in Figs. 8 and 9 is valid.

2.3 Manual Quantifier Instantiation.

To achieve decidable verification queries, Ravencheck employs the well-known *relational abstraction* technique to eliminate problematic function symbols [23]. From the user's perspective, this technique manifests as a form of incompleteness: counterexamples may be found for valid properties. To help users diagnose and repair this incompleteness, Ravencheck presents a *partial function* semantics: the solver is permitted to falsify a condition by assuming that `#[declare]` functions are undefined for inputs that do not appear within the condition itself. A formal presentation of Ravencheck's partial function semantics is given in [17]. As an example, consider the `factor_s_right` property shown below, which attempts to prove that $add(a, S(b)) == S(add(a, b))$. Unlike in the previous inductive verification example, `add` is treated here as a fully *uninterpreted function*, meaning the solver knows nothing about its behavior beyond what is given in axioms. The goal is to prove that $add(a, S(b)) == S(add(a, b))$. Although this follows logically from the given commutativity and factoring axioms, the SMT solver fails to verify it. Under Ravencheck's partial-function semantics, an axiom is applicable only when its terms are fully defined. If any term within an axiom instance can be undefined, the axiom itself short-circuits to `true`, preventing it from placing any constraint on the solver. Consequently, unless the function call term (e.g., `add(S(b), a)`) explicitly appears in the verification condition, the solver can ignore the body of `add_commute` for values `add(S(b))` and `a` when producing a counterexample.

To address this, the user can add trivial statements like `let _ = add(S(b),a);` to the verification condition, which forces the solver to recognize that `add(S(b), a)` is defined. This fix is an example of *quantifier instantiation*, a well-known solution to relational abstraction incompleteness [23]. Ravencheck makes this technique accessible to developers by presenting it in terms of familiar programming lanuguage constructs. It is important to note that this scenario illustrates a technique that users can employ in edge cases. Many verification problems can be solved in Ravencheck without resorting to this technique, including the inductive version of the `factor_s_right` property in Sec. 2.2 and the more complex properties detailed in [17]. In fact, we found that quantifier instantiations were only required for a small fraction of benchmarks in one of the three case studies conducted in [17]. The `#[total]` attribute can be used on functions like `get` in Fig. 3 to force the solver to recognize them as total. Functions like `set` and `add` unfortunately must be partial, because their inputs and outputs share the same types: total functions like fall outside the Extended EPR fragment.

```
#[verify]
fn swap_set() -> bool {
    forall(|h:Heap, x:Ptr, y:Ptr, v:Ptr| {
        let h1 = set(h, x, v);
        get(swap(h1, x, y), y) == v
    })
}
```

Figure 11. Verification condition on `get`, `set`, and `swap`.

```
forall(|h, x, y, v| {
  let h0 = set(h, x, v);
  let v0 = get(h0, x);
  let v1 = get(h0, y);
  let h1 = set(h0, x, v1);
  let h2 = set(h1, y, v0);
  let v2 = get(h2, y);
  v2 == v
}
```

Figure 12. The VC from Fig. 11 after partial evaluation.

3 Internal Representation and Encoding

To perform verification, Ravencheck lowers the input Rust code through a multi-stage pipeline to produce an SMT formula that satisfies the Extended EPR fragment. The pipeline consists of three key phases: (1) lowering Rust syntax into the Ravencheck IR, (2) partial evaluation into a logical expression, and (3) relational abstraction to replace function symbols. We demonstrate this process the example property in Fig. 11, using the `get`, `set`, and `swap` functions that we defined in Sec. 2.

Lowering to the IR.. The first stage converts the surface-level Rust AST into an intermediate representation (IR) based on Call-By-Push-Value (CBPV) [15]. This process rejects Rust syntax features that we do not support in logical formulas, such as mutation statements and method calls.

Partial Evaluation. Once in the IR, the condition is partially evaluated to remove programming indirection constructs such as defined functions and nested let-bindings.

$$
\begin{aligned}
&\forall h, x, y, v.\\
&\quad \forall h_0.\ R_{set}(h, x, v) \Rightarrow\\
&\quad \forall v_0.\ R_{get}(h_0, x) \Rightarrow\\
&\quad \forall v_1.\ R_{get}(h_0, y) \Rightarrow\\
&\quad \forall h_1.\ R_{set}(h_0, x, v_1) \Rightarrow\\
&\quad \forall h_2.\ R_{set}(h_1, y, v_0) \Rightarrow\\
&\quad \forall v_2.\ R_{get}(h_2, y) \Rightarrow\\
&\quad v_2 = v
\end{aligned}
$$

Figure 13. Relational abstraction of Fig. 12.

The resulting expression is a logical formula in A-normal form, in which functions are only called on variables and constants. Interpreted functions (annotated with `#[define]` in the source) are inlined during this process, so that only uninterpreted (`#[declare]`) functions remain. In Fig. 12, the `swap` call has been inlined, so that only `set` and `get` remain.

Relational Abstraction. Functions like `get` and `set` introduce sort cycles [12] since one of their argument types is same as their return type. To accommodate the use of such functions while still targeting the decidable Extended EPR fragment, we replace all function symbols with relational abstractions. Each call to an uninterpreted function in Fig. 12 is replaced by a implication, which quantifies the function's output and assumes that it is related to its input by an uninterpreted relation symbol, as shown in Fig. 13.

4 Example: Verifying Heap Reachability

We demonstrate Ravencheck's applicability to complex, stateful program analysis via a verified linked-list reversal algorithm. Analyzing heap-manipulating programs is traditionally challenging, and Ravencheck addresses this by modeling heap mutations as externally-pure functions, within tail-call-optimized recursive functions.

Externally-Pure State Mutation. As shown in Fig. 14, the state is modeled using a `Heap` representing an array of pointers. The primitive `set_next` updates a pointer. Since it is declared as an uninterpreted function in Ravencheck, its body can contain code with mutable updates. While returning a completely new heap appears computationally expensive, the `Heap` type is actually a unique reference (`Box`) to a heap-allocated array. At runtime, the body of `set_next` simply mutates a single cell in that array in-place and passes the unique reference back. Rust's borrow-checker ensures the old heap reference is not reused, preventing aliasing issues. Furthermore, the Rust compiler will inline the `set_next` call, eliminating any function-call overhead.

Verification and Compilation. We axiomatize the heap domain using a `reach(x, y, h)` relation, which denotes that a path of connected pointers exists from `x` to `y` within the heap `h`. Our goal specification for a reversed list asserts that a path exists in the input list if and only if the flipped path exists in the output list. To achieve this, we use `for_root` to universally quantify a terminal root pointer, so that the property is only required to hold when the input list is acyclic. Since the Rust compiler rarely performs automatic tail-call optimization, the developer cannot directly define `reverse` as a recursive function—rather, they define the components (`init`, `cond`, `step`, and `finish`). Ravencheck automatically assembles these into two semantically equivalent—but syntactically distinct—forms. The first is a recursive function used for verification, and the second is its tail-call optimization (a loop), passed onto the Rust toolchain for compilation. This allows the verified code to achieve runtime performance indistinguishable from unsafe imperative code.

```
#[declare] type Heap = Box<[usize; HEAP_MAX]>;
#[declare] struct Ptr(usize);
#[declare] type Opt = Option<Ptr>;

#[declare] fn set_next(Ptr(x): Ptr, y: Opt, mut h: Heap)
     -> Heap {
  match y {
    Some(Ptr(y)) => { h[x] = y; h },
    None => { h[x] = NULL_PTR; h }
  }
}

// Loop Components
fn init(c: Opt, h: Heap) -> State { (c, None, h) }
fn cond((c, _, _): State) -> Opt { c }
fn step(c: Ptr, (_, d, h): State) -> State {
    let c2 = get_next(c, &h);
    let h2 = set_next(c, d, h);
    (c2, Some(c), h2)
}
fn finish((_, d, h): State) -> (Opt, Heap) { (d, h) }

#[verify] fn reverse_is_safe() -> bool {
  forall(|co: Ptr, ho: Heap, df: Ptr, hf: Heap| {
    forall(|x: Ptr, y: Ptr| { for_root(co, ho, |root:
         Ptr| {
      (reach(df,y,hf) && reach(y,x,hf))
        == (reach(co,x,ho) && reach(x,y,ho))
    }) })
  })
}
```

Figure 14. Verifying heap manipulations via pure functions.

5 Conclusion

Ravencheck addresses the challenge of proof instability in systems software verification by grounding its logic in the decidable EPR fragment. Distinctively, it integrates seamlessly with the standard toolchain, eliminating the need for external modeling languages. Ravencheck allows developers to write verification conditions using the same purely functional subset of Rust used for executable code. By unifying specification and implementation without sacrificing runtime performance and decidability, Ravencheck establishes a pragmatic foundation for building verified systems in Rust.

Acknowledgements. We thank to CUPLV members and anonymous reviewers for thoughtful feedbacks. This project was partially funded by the NSF FMitF award #2422136.

References

[1] Vytautas Astrauskas, Aurel Bílý, Jonáš Fiala, Zachary Grannan, Christoph Matheja, Peter Müller, Federico Poli, and Alexander J Summers. 2022. The prusti project: Formal verification for rust. In *NASA Formal Methods Symposium*. Springer, 88–108.

[2] Haniel Barbosa, Clark Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, et al. 2022. cvc5: A versatile and industrial-strength SMT solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 415–442.

[3] Yves Bertot and Pierre Castéran. 2013. *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions*. Springer Science & Business Media.

[4] Koen Claessen, Moa Johansson, Dan Rosén, and Nicholas Smallbone. 2015. TIP: Tons of Inductive Problems. In *Intelligent Computer Mathematics*, Manfred Kerber, Jacques Carette, Cezary Kaliszyk, Florian Rabe, and Volker Sorge (Eds.). Springer International Publishing, Cham, 333–337.

[5] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 337–340.

[6] Leonardo De Moura, Soonho Kong, Jeremy Avigad, Floris Van Doorn, and Jakob von Raumer. 2015. The Lean theorem prover (system description). In *International Conference on Automated Deduction*. Springer, 378–388.

[7] Xavier Denis, Jacques-Henri Jourdan, and Claude Marché. 2022. Creusot: A foundry for the deductive verification of Rust programs. In *International Conference on Formal Engineering Methods*. Springer, 90–105.

[8] Jean-Christophe Filliâtre, Léon Gondelman, and Andrei Paskevich. 2016. A pragmatic type system for deductive verification. (2016).

[9] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R Lorch, Bryan Parno, Michael L Roberts, Srinath Setty, and Brian Zill. 2015. IronFleet: proving practical distributed systems correct. In *Proceedings of the 25th Symposium on Operating Systems Principles*. 1–17.

[10] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2017. RustBelt: securing the foundations of the Rust programming language. *Proc. ACM Program. Lang.* 2, POPL, Article 66 (Dec. 2017), 34 pages. doi:10.1145/3158154

[11] Steve Klabnik and Carol Nichols. 2023. *The Rust programming language*. No Starch Press.

[12] Konstantin Korovin. 2013. Non-cyclic Sorts for First-Order Satisfiability. In *Frontiers of Combining Systems*, Pascal Fontaine, Christophe Ringeissen, and Renate A. Schmidt (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 214–228.

[13] Andrea Lattuada, Travis Hance, Jay Bosamiya, Matthias Brun, Chanhee Cho, Hayley LeBlanc, Pranav Srinivasan, Reto Achermann, Tej Chajed, Chris Hawblitzel, Jon Howell, Jacob R. Lorch, Oded Padon, and Bryan Parno. 2024. Verus: A Practical Foundation for Systems Verification. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles* (Austin, TX, USA) *(SOSP '24)*. Association for Computing Machinery, New York, NY, USA, 438–454. doi:10.1145/3694715.3695952

[14] K Rustan M Leino. 2010. Dafny: An automatic program verifier for functional correctness. In *International conference on logic for programming artificial intelligence and reasoning*. Springer, 348–370.

[15] Paul Blain Levy. 2013. *Call-by-push-value*. Ph. D. Dissertation.

[16] Nicholas V. Lewchenko, Gowtham Kaki, and Bor-Yuh Evan Chang. 2025. Bolt-On Strong Consistency: Specification, Implementation, and Verification. *Proc. ACM Program. Lang.* 9, OOPSLA1, Article 137 (April 2025), 28 pages. doi:10.1145/3720502

[17] Nicholas V Lewchenko, Kunha Kim, Bor-Yuh Evan Chang, and Gowtham Kaki. 2026. Effectively Propositional Higher-Order Functional Programming. *Proceedings of the ACM on Programming Languages* 10, OOPSLA1 (2026), 1627–1653.

[18] Chris Okasaki. 1998. *Purely functional data structures*. Cambridge University Press.

[19] Oded Padon, Kenneth L McMillan, Aurojit Panda, Mooly Sagiv, and Sharon Shoham. 2016. Ivy: safety verification by interactive generalization. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 614–630.

[20] Lawrence C Paulson. 1994. *Isabelle: A generic theorem prover*. Springer.

[21] Project Everest Team. 2025. Project Everest: Perspectives from Developing Industrial-grade High-Assurance Software. *ACM Transactions on Programming Languages and Systems (TOPLAS)* (May 2025). https://project-everest.github.io/assets/everest-perspectives-2025.pdf To appear.

[22] Nikhil Swamy, Guido Martinez, and Aseem Rastogi. 2023. Proof-Oriented Programming in F*.

[23] Orr Tamir, Marcelo Taube, Kenneth L. McMillan, Sharon Shoham, Jon Howell, Guy Gueta, and Mooly Sagiv. 2023. Counterexample Driven Quantifier Instantiations with Applications to Distributed Protocols. *Proc. ACM Program. Lang.* 7, OOPSLA2, Article 288 (Oct. 2023), 27 pages. doi:10.1145/3622864

[24] R Kent Treiber et al. 1986. *Systems programming: Coping with parallelism*. International Business Machines Incorporated, Thomas J. Watson Research ....

[25] Niki Vazou, Eric L Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon Peyton-Jones. 2014. Refinement types for Haskell. In *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming*. 269–282.

[26] Yi Zhou, Jay Bosamiya, Yoshiki Takashima, Jessica Li, Marijn Heule, and Bryan Parno. 2023. Mariposa: Measuring SMT instability in automated program verification. In *2023 Formal Methods in Computer-Aided Design (FMCAD)*. IEEE, 178–188.