Effectively Propositional Higher-Order Functional Programming

NICHOLAS V. LEWCHENKO, University of Colorado Boulder, USA
KUNHA KIM, University of Colorado Boulder, USA
BOR-YUH EVAN CHANG*, University of Colorado Boulder, USA and Amazon, USA
GOWTHAM KAKI, University of Colorado Boulder, USA

Decidable automation is a key feature of program verification tools, which makes them easier for non-expert developers to use and understand. Unfortunately, decidable fragments of logic are very restrictive, and not ideal for the expression of idiomatic programs. The decidable *Extended EPR* fragment, combined with an encoding technique called *relational abstraction*, has seen wide use for its ability to handle both quantifiers and uninterpreted functions. However, the complexity of the relational abstraction encoding, combined with its inherent incompleteness, still poses a significant obstacle to non-experts.

In this paper, we show that Extended EPR and relational abstraction can be deployed to achieve decidable, quantified verification within a familiar, principled domain: a higher-order, purely-functional programming language. We observe that, by defining the semantics of our language in terms of partial functions, we obtain a well-behaved, three-valued logic that matches the behavior of the relational abstraction encoding while hiding its complexity. We demonstrate that our prototype implementation can ergonomically replicate EPR-based program verification benchmarks and verify recursive programs on inductive datatypes.

CCS Concepts: • **Software and its engineering → Formal software verification**; • **Theory of computation** → *Automated reasoning*.

Additional Key Words and Phrases: Decidable Verification, Relational Abstraction, Three-Value Logic

ACM Reference Format:
Nicholas V. Lewchenko, Kunha Kim, Bor-Yuh Evan Chang, and Gowtham Kaki. 2026. Effectively Propositional Higher-Order Functional Programming. *Proc. ACM Program. Lang.* 10, OOPSLA1, Article 151 (April 2026), 27 pages. https://doi.org/10.1145/3798259

1 Introduction

Functional programming languages, such as OCaml and Haskell, are known to prioritize developer productivity and software reliability. An oft-cited virtue of functional languages is that they are amenable to equational reasoning and formal verification [Hudak 1989; Pierce 2002; Wadler 1992], which makes them an ideal medium to express mathematical proofs in interactive theorem provers, such as Rocq, Lean, and Isabelle. While proof assistants have witnessed increased adoption in recent years, their utility towards functional program verification itself remains limited, primarily due to the prohibitive manual effort involved. A promising approach to reduce the manual effort

*Bor-Yuh Evan Chang holds concurrent appointments at the University of Colorado Boulder and as an Amazon Scholar. This paper describes work performed at the University of Colorado Boulder and is not associated with Amazon.

Authors' Contact Information: Nicholas V. Lewchenko, University of Colorado Boulder, Boulder, USA, nicholas.lewchenko@colorado.edu; Kunha Kim, University of Colorado Boulder, Boulder, USA, kunha.kim@colorado.edu; Bor-Yuh Evan Chang, University of Colorado Boulder, Boulder, USA and Amazon, Seattle, USA, evan.chang@colorado.edu; Gowtham Kaki, University of Colorado Boulder, Boulder, USA, gowtham.kaki@colorado.edu.

ACM 2475-1421/2026/4-ART151
https://doi.org/10.1145/3798259

is to implement proof automation with help of SMT solvers, such as Z3 and CVC5. While SMT-aided formal verification is not *foundational* – it assumes a large trusted codebase and does not generate a machine-checkable proof object – it has been proven to be quite effective at establishing the safety of real-world software. A more serious limitation of the SMT-aided approach is its *unpredictability* resulting from the fundamental undecidability of first-order logic (FOL) in which program verification problems are generally encoded. While SMT solvers are decision procedures for propositional logic and its derivatives, they rely on complex heuristics to decide satisfiability of quantified formulas. The heuristics interact in non-trivial ways, making their behavior on complex verification conditions hard to predict and debug. Indeed, it is well-known that SMT-aided verification frameworks, such as Dafny [Leino 2010] and F* [Swamy 2023], suffer from *proof instability*: small, semantically irrelevant changes can cause verification to fail [Zhou et al. 2023].

We can avoid proof instability in a verification framework by restricting the verification conditions (VCs) to quantifier-free fragments of formal logic for which SMT solvers are decision procedures, e.g., Linear Integer Arithmetic (LIA). If a program cannot be encoded in a quantifier-free fragment, its proof needs to be spelled-out by explicitly constructing a proof object. SMT-aided dependent type-checking frameworks, such as Liquid Haskell [Blanchette et al. 2022; Vazou et al. 2014] and F* [Swamy 2023], come with a library of metaprogramming *tactics* to help developers build such proof objects. Liquid Haskell restricts the type refinements to LIA with uninterpreted functions. Quantified assertions are expressed as functions with existential dependent types for which proof objects need to be built by the programmer. While this approach reduces the manual effort compared to foundational verification, it makes no use of SMT's formidable infrastructure to decide quantified assertions, choosing to rely on the programmer to spell out the proof.

There exist several fragments of first-order logic that are decidable [Börger et al. 1996]. The Effectively Propositional (EPR) fragment together with its extension to stratified functions [Korovin 2013] is one such fragment that has proven to be useful in formal verification problems across diverse domains [Itzhaky et al. 2014a; Kaki and Jagannathan 2014; Padon et al. 2017]. Extended EPR is *effectively propositional*: although the logic includes quantifiers, it restricts nested quantification to make it possible to *ground* assertions into a propositional form. The restrictions on quantification excludes theories, such as LIA, and basic operations, such as integer addition and set union. In practice, such operators are approximated as uninterpreted boolean functions, i.e., relations. This so called *relational abstraction* is incomplete as it overapproximates functions, resulting in spurious counterexamples. To ensure predictable automation, care must be taken to ensure that an EPR encoding of a program verification problems reflects its intended semantics. To accommodate this necessity, Ivy [McMillan and Padon 2020] defines a bifurcated, first-order language with distinct sub-languages for quantified logical expressions and (relationally-abstracted) imperative *actions*. Ivy is primarily a modeling language for state transition systems, and actions are meant to represent non-deterministically invoked transitions that update the state. Actions are therefore not functions and it is unclear how they fit the domains that do not involve imperative aspects and non-determinism. Integrating EPR-backed decidable verification into a mainstream functional programming language with well-defined semantics remains an open question.

In this paper, we show that the relationally-abstracted Extended EPR approach to verification can be made *predictable*—it can be packaged beneath a simple surface language with straightforward semantics. We are inspired by the interface of *property-based testing* frameworks, in which property assertions are expressed directly in the implementation language that the developer is already familiar with. Following this example, we present Λ_{EPR}, a higher-order functional programming language with quantified assertions and decidable verification. Λ_{EPR} provides logical quantifiers directly alongside the key mechanism of functional programming—higher-order functions—and is thus simultaneously fit for the definition of programs and the definition of quantified program

assertions. Furthermore, Λ_{EPR} can be interpreted as a fragment of higher-order logic (HOL), giving it a straightforward formal semantics amenable to easy interpretation.

Decidable verification for Λ_{EPR} depends on three key innovations. The first is an encoding from Λ_{EPR} into SMT-representable first order logic, which eliminates higher-order functions and replaces primitive operators with relational abstractions. The second is a type system that tracks the nesting of logical quantifiers and operators, disallowing interactions that would take the FOL encoding outside of Extended EPR while keeping programs composable. The third is a formal denotational semantics that is sound and complete with respect to the encoded verification conditions. While functions in Λ_{EPR}—primitive operators and user-defined functions—are, by definition, total, we interpret their semantics in terms of *partial functions* in order to precisely capture the relational abstraction incompleteness. When only total functions are considered, this semantics matches the standard semantics of HOL, making our verification sound with respect to a commonly-understood baseline. To our knowledge, Λ_{EPR} is the first programming language that supports automated verification based on HOL, in contrast to existing languages based on dependent type theory.

We have implemented Λ_{EPR} as a verification tool for the Rust programming language, called Ravencheck. Users write executable programs and verification conditions in a purely-functional subset of Rust, which are translated into Λ_{EPR} for verification. The programs can then be compiled by the standard Rust toolchain for efficient runtime execution, and combined with non-verified Rust code for integration into existing codebases. We have published Ravencheck as an open-source Rust crate [Lewchenko and Contributors 2026], which can be integrated as a dependency into any existing Rust project.

Contributions. We summarize our contributions below:

- We present Λ_{EPR}, a fragment of higher-order logic that serves as a functional programming language while supporting SMT-aided decidable verification. Λ_{EPR} comes with a type system to enforce decidability of verification conditions, but is *not* dependently typed. It does not differentiate between expression and assertion languages and offers an interface similar to property-based testing frameworks (Sec. 2-3).
- We define a denotational semantics for Λ_{EPR} in terms of partial functions that captures the semantics of its encoding in the extended EPR fragment without having to expose the low-level details. We show that this semantics is sound with respect to the standard HOL semantics, but necessarily incomplete to ensure decidability (Sec. 4).
- We define an encoding of Λ_{EPR} in the Extended EPR fragment and show that the encoding is sound with respect to the partial function semantics, and complete in the absence of abstracted higher-order functions (Sec. 5).
- Using our Ravencheck verification tool, we demonstrate the effectiveness of Λ_{EPR} in case-studies across three domains: inductive datatype problems from the TIP benchmarks [Claessen et al. 2015], heap reachability for pointer-manipulating programs [Itzhaky et al. 2013], and distributed consensus algorithms [Ongaro and Ousterhout 2014] (Sec. 6).

2 Motivation and Overview

In this section, we present the motivation behind our decidable programming language, Λ_{EPR}, and its several design choices. The language Λ_{EPR} and the associated verification tool, Ravencheck, are both implemented in Rust, so we use Rust syntax in this section.

2.1 Example: Distributed Coordination

We begin with a small fragment of a distributed coordination system: a round of voting by nodes in a distributed system to elect a new leader. The key safety property is *exclusivity*: there can never be

```
type Votes = Set<(Pid,Pid)>;    fn init() -> Votes { set_empty }
fn has_voted(p: Pid, votes: Votes) -> bool {
    let p_votes = filter(|(v,_)| v == p, votes);
    p_votes /= empty_set }
fn add_vote(v: Pid, c: Pid, votes: Votes) -> Votes {
    if !has_voted(v, votes) { insert((v,c), votes) } else { votes } }
fn one_vote(votes: Votes) -> bool {
    forall(|p1: Pid, p2: Pid, p3: Pid| {
        member((p1,p2), votes) && member((p1,p3), votes) => p2 == p3 })}
fn is_invariant(f: impl Fn(Votes) -> bool) -> bool {
    f(init) && forall(|s: Votes| { f(s) => forall(|(v,c)| f(add_vote(v,c,s))) })}
#[verify] fn safety() -> bool { is_invariant(one_vote) }
```

Fig. 1. A code snippet implementing the voting functionality of a simple leader election protocol. We assume the set functions `member`, `filter`, `empty_set`, and `insert` are provided by an imported library.

two votes cast by a single node. Fig. 1 shows a code snippet of the leader election protocol together with its specification and the verification condition. In our implementation of Λ_{EPR}, executable functions are written in a subset of Rust, and logical expressions extend this subset with the `forall` quantifier. Intuitively, the code manipulates a `Votes` state: a set of vote pairs (v, c), where v is the voter and c is the candidate.

The specification of the leader election protocol is expressed as the invariant `one_vote`, which requires that, for a given state, two distinct votes cannot exist that share the same voter. To make this invariant inductive, we add the condition `one_vote`: a state cannot contain two different votes from the same participant. The verification condition, named `safety`, checks that `one_vote` is indeed an invariant, inductive over any application of `add_vote`, starting from an `init` state.

Verification. Verification requires axiomatizations of library functions, such as `member` and `insert`. Similar to F* [Swamy 2023], we assume that such axioms are exposed through the library interface. For example, the axiom for `insert` is as follows:

$$A_{\text{ins}} \triangleq \forall x_1, x_2, s_1.\, \texttt{member}\, x_2\, (\texttt{insert}\, x_1\, s_1) \Leftrightarrow (\texttt{member}\, x_2\, s_1 \vee x_2 = x_1)$$

Using the axiomatizations of library functions, one can manually verify the leader election protocol in an interactive proof assistant, e.g., Isabelle. *Predictably* automating verification via an SMT solver, however, requires more work as it requires us to encode the verification condition (VC) into a decidable fragment of formal logic. The Extended EPR fragment [Korovin 2013] is a natural choice considering that this example involves quantifiers. Extended EPR extends the EPR fragment with quantifier alternation and uninterpreted functions insofar as the dependency graph of *sorts*, i.e., types, involved in these extensions does not form a cycle. The dependency graph is computed by adding an edge $s_2 \longrightarrow s_1$ for every: (a) quantifier alternation of form $\forall(x_1 : s_1) \ldots \exists(x_2 : s_2)$, and (b) function whose sort is of form $(\ldots \times s_1 \times \ldots) \longrightarrow s_2$.

Unfortunately, the current example contains sort cycles: functions `add_vote` and `insert` take `Set<(Pid,Pid)>` as both an input and output, which induces a self loop in the dependency graph. While the function symbol for `add_vote` can be eliminated by inlining its definition in the VC, the same cannot be said of the imported function `insert` whose definition is unavailable.

Fortunately, we can eliminate the sort cycle induced by `insert` by replacing it with its *relational abstraction* R_{ins}. R_{ins} can be understood as an underapproximation of `insert`, such that $\forall x, s_1, s_2.\, R_{\text{ins}}(x, s_1, s_2) \implies \texttt{insert}\, x\, s_1 = s_2$. This lets us rewrite the axiom A_{ins} in terms of R_{ins}:

$$A^R_{\text{ins}} \triangleq \forall x_1, x_2, s_1, s_2.\, R_{\text{ins}}(x_1, s_1, s_2) \implies \texttt{member}\, x_2\, s_2 \Leftrightarrow (\texttt{member}\, x_2\, s_1 \vee x_2 = x_1)$$

Assuming $R_{\texttt{ins}}$ is an underapproximation of `insert`, we have $A_{\texttt{ins}} \Rightarrow A^R_{\texttt{ins}}$. Now, let us consider a first-order property Φ_1 on `insert` that is valid under $A_{\texttt{ins}}$, i.e., $A_{\texttt{ins}} \Rightarrow \Phi_1$. Let Φ^R_1 denote the formula obtained by replacing the occurrences of `insert` with a quantified assumption of $R_{\texttt{ins}}$, as in the formula given above. The key observation we make is that for specific choices of Φ_1, $A_{\texttt{ins}} \Rightarrow \Phi_1$ is valid *if and only if* $A^R_{\texttt{ins}} \Rightarrow \Phi^R_1$ is valid in first-order logic. To demonstrate, consider the following (invalid) assertion on `insert`:

$$\Phi_1 \quad \triangleq \quad \forall x_1, x_2, s_1.\ \texttt{member}\ x_2\ (\texttt{insert}\ x_1\ s_1)$$

The corresponding relationally abstracted version is as follows:

$$\Phi^R_1 \quad \triangleq \quad \forall x_1, x_2, s_1, s_2.\ R_{\texttt{ins}}(x_1, s_1, s_2) \ \Rightarrow\ \texttt{member}\ x_2\ s_2$$

Observe that any counterexample $\mathcal{M}_1$ for $A_{\texttt{ins}} \Rightarrow \Phi_1$ can be transformed into a counterexample $\mathcal{M}^R_1$ for $A^R_{\texttt{ins}} \Rightarrow \Phi^R_1$: since $\mathcal{M}_1$ is a model of $A_{\texttt{ins}} \wedge \neg\Phi_1$, its universe contains three values, S_1, S_2, and X_1, such that $S_2 = \texttt{insert}\ X_1\ S_1$. $\mathcal{M}^R_1$ is obtained by adding the $R_{\texttt{ins}}$ relation symbol to $\mathcal{M}_1$ and setting $R_{\texttt{ins}}(X_1, S_1, S_2) = \texttt{True}$. Note that the converse is also true: any counterexample $\mathcal{M}^R_2$ for Φ^R_1 can be converted into a counterexample $\mathcal{M}_2$ for Φ_1: since $\mathcal{M}^R_2$ is a model of $A^R_{\texttt{ins}} \wedge \neg\Phi^R_1$, it has to set Φ^R_1 to `False`, which can only be done by setting $R_{\texttt{ins}}(X_1, S_1, S_2)$ to `True` for the values X_1, S_1, and S_2. We can now build $\mathcal{M}_2$ by adding the `insert` function symbol to $\mathcal{M}^R_2$ and setting $\texttt{insert}\ X_1\ S_1 = S_2$. In summary, for the particular property Φ_1, every counterexample of $A_{\texttt{ins}} \Rightarrow \Phi_1$, corresponds to a counterexample for $A^R_{\texttt{ins}} \Rightarrow \Phi^R_1$, and vice-versa. Thus, at least for Φ_1, we can rely on an SMT solver to check validity.

2.2 Partial-Function Semantics

Unfortunately relational abstraction is not always semantics preserving. For example:

$$\Phi_2 \quad \triangleq \quad \forall x.\exists s.\ \texttt{member}(x, s)$$

The formula $A_{\texttt{ins}} \Rightarrow \Phi_2$ is considered valid as no non-empty models exist for $A_{\texttt{ins}} \wedge \neg\Phi_2$. Since Φ_2 refers only to the relation `member` and no functions, it is its own relational abstraction. The relationally abstracted version of $A_{\texttt{ins}} \Rightarrow \Phi_2$ is therefore $A^R_{\texttt{ins}} \Rightarrow \Phi_2$. This is valid iff $A^R_{\texttt{ins}} \wedge \neg\Phi_2$ is unsatisfiable. The latter is satisfiable as there exists a non-empty model $\mathcal{M}$ that defines constants X_1, S_1 and S_2 and interprets relations $R_{\texttt{ins}}$ and `member` as empty sets. $\mathcal{M}$ cannot be a model of $A_{\texttt{ins}} \Rightarrow \Phi_2$ as an empty set cannot serve as the model for function `insert`. One may attempt to fix this by asserting a *totality* axiom for $R_{\texttt{ins}}$ that preempts an empty set:

$$\texttt{totality} \quad \triangleq \quad \forall x, s_1.\exists s_2.\ R_{\texttt{ins}}(x, s_1, s_2)$$

Unfortunately, the $\forall s_1.\exists s_2$ quantifier alternation adds a self loop on `set` sort, excluding it from Extended EPR. There is indeed no automatic fix to align the relational abstraction Φ^R with the original semantics of the formula Φ. Incompleteness can be manually repaired using quantifier instantiations (Sec. 2.6), but diagnosing the cause requires understanding the details of the relational-abstraction encoding. To enable developers to use EPR verification without diving into such details, we propose a weaker semantics for the original Φ, under which validity matches that of Φ^R.

Our approach uses higher-order logic (HOL) with *partial functions* as a means to understand the semantics of the EPR-constrained verification problem associated with a higher-order functional program. Higher-order functional programs have a natural interpretation in HOL[1] when all functions are total. Given the HOL interpretation $[\![P]\!]$ of a functional program P, the verification problem asks if $\Phi([\![P]\!])$ is valid in all models where the function symbols are interpreted as total functions. Our key observation is that the relational abstraction of $\Phi([\![P]\!])$ is sound and complete with respect to the semantics of $\Phi([\![P]\!])$ in HOL when partial-function models are admitted. Thus,

[1]We ignore termination concerns for the time being.

to determine whether an assertion failure is caused by relational abstraction incompleteness, one only needs to consider the assertion under the possibility that uninterpreted functions are partial.

We accommodate partial functions in HOL using three-valued logic. For illustration, let us reconsider Φ_1 under partial-function semantics, considering a model where `insert` is undefined for some inputs. Consequently, the expression `member` x_2 (`insert` x_1 s_1) is `Undefined` for some choices of x_1 and s_1. When `insert` is defined, however, the body has either a `True` or `False` value. Our semantics defines $\forall x.\ \Phi$ to be `False` when any choice of x makes Φ `False`. Therefore, our semantics considers Φ_1 to be `False` if `member` x_2 (`insert` x_1 s_1) can be false for any defined output of `insert`, regardless of cases in which `insert` is not defined. When Φ is `True` for some cases and `Undefined` for others, our semantics defines $\forall x.\ \Phi$ to be `Undefined`. Therefore, if `member` x_2 (`insert` x_1 s_1) is never `False` for any defined output of `insert`, then the value of Φ_1 will also never be false.

Essentially, while the encoded FOL assertion always has the value `True` or `False`, our partial-function HOL semantics replaces some of those `True` results with `Undefined`. We consider an assertion *valid*—and expect verification to succeed—if and only if every model gives it a `True` *or* `Undefined` value in the partial-function HOL semantics. Therefore, we can judge Φ_1 to be valid and expect verification for it to succeed.

2.3 Quantifier Interaction Types

We've seen how functions can break our decidability guarantee, and how we avoid this using relational abstraction. Unfortunately, the Extended EPR fragment also restricts our use of quantifiers: an assertion cannot include nested quantifiers of the form $\exists x.\ \forall y.\ \varphi(x, y)$ when x and y have the same type. In fact, more generally, we must avoid any *cycle* of types. So we can't have $(\exists x : A.\ \forall y : B.\ \varphi_1(x, y)) \wedge (\exists y : B.\ \forall x : A.\ \varphi_2(x, y))$ either.

As it happens, our example in Fig. 1 uses its quantifiers in such a way that no $\exists\forall$ connections between types are created at all. But maybe this is just a lucky case—it's difficult, when you consider negation and function application, to even see exactly which alternations occur.

To assist developers with maintaining the decidability of their verification conditions, we have added *quantifier interaction* information to the type system of Λ_{EPR}. For example, the type of `one_vote s` includes the *quantifier interaction tree* $\{\forall \texttt{Pid}.\emptyset\}$, indicating that it universally quantifies one (or more) `Pid`s, with no further quantifiers underneath ($\emptyset$). Negation flips the elements of that tree: the term `not (one_vote s)`, has the tree $\{\exists \texttt{Pid}.\emptyset\}$.

Consider the term `is_invariant(one_vote)`, which appears in the `safety` condition. This term is assigned the following tree:

$$\{\forall \texttt{Votes}.\ \{\exists \texttt{Pid}.\emptyset,\ \forall \texttt{Pid}.\ \{\forall \texttt{Pid}.\ \{\mathcal{F}\,\texttt{Votes}.\emptyset\}\}\}\}$$

At the top level of `is_invariant`, we $\forall$-quantify the `Votes` value `s`. Beneath this is an implication: the operand `one_vote(s)` is negated (being on the left side of the implication), and so an $\exists$`Pid`.$\emptyset$ tree is nested as a child of the top-level $\forall$`Votes`. The right (positive) operand $\forall$-quantifies more `Pid`(s), and then applies `one_vote(add_vote(v, c, s))`. This instance of `one_vote` creates one more layer of $\forall$`Pid` quantification, and then finally `add_vote` adds the *functional quantifier* node $\mathcal{F}$`Votes`.$\emptyset$. Specifically, this node is due to the use of `insert` within `add_vote`, which is an irreducible imported function with output type `Votes`.

The application of imported functions is tracked in the quantifier interaction tree because, in our relational abstraction encoding, these applications are replaced with guarded universal quantifiers. Thus, the $\mathcal{F}$`Votes` node behaves mostly like a $\forall$`Votes` node, with one difference: unlike true $\forall$ nodes, negation does not flip an $\mathcal{F}$ to an $\exists$.

To determine whether an assertion falls within Extended EPR, we construct a *sort graph* from its quantifier interaction tree. Each non-boolean base type (i.e. sort), such as `Pid` or `Voters`, forms

a node in the graph. We have a directed edge from `Pid` to `Voters` iff a path in the quantifier interaction tree takes you from an $\exists$`Pid` node to either a $\forall$`Voters` node or an $\mathcal{F}$`Voters` node.

To fully determine decidability, we must merge the sort graph from our assertion with that generated by our axioms (`assume` statements). Because axioms occur in negative position, edges are generated by the opposite rules: a path from a $\forall D_1$ node to either an $\exists D_2$ or an $\mathcal{F} D_2$ node creates a $D_1 \longrightarrow D_2$ edge.

In the leader election example, an axiom for the `is_quorum` predicate on voter-sets is notable:

```
#[assume] fn quorum_intersect() -> bool { forall(|s1,s2,s3: Voters| {
    is_quorum(s1,s3) && is_quorum(s2,s3)
        ==> exists(|p: Pid| member p s1 && member p s2 )})}
```

`is_quorum` checks whether its left argument contains a majority of the members of the right argument. This axiom has quantifier tree $\forall$`Voters`. $\exists$`Pid`.$\emptyset$, and thus generates a `Voters` $\longrightarrow$ `Pid` edge in the overall sort graph. This means that our assertion must not generate the opposing `Pid` $\longrightarrow$ `Voters` edge.

Function types. A function's argument can influence its quantifier interaction tree. We handle this in the type system by assigning functions *abstract* quantifier trees, which take an argument. For example, consider the tree for the unapplied `is_invariant` function:

$$\texttt{is\_invariant} : \Lambda\alpha.\ \{\forall \texttt{Votes}.\ \{\neg\alpha(\emptyset),\ \forall \texttt{Pid}.\ \{\alpha(\mathcal{F}\texttt{Votes}.\emptyset)\}\}\}$$

In this tree, α stands in for the argument's tree, which becomes a subtree in two places—corresponding to the two places that `is_invariant`'s argument is used. Note that α itself is an abstract tree, since `is_invariant`'s argument is a function. On the left side of the implication, the argument is applied to `s` and is in negative position, so we apply its tree to the empty tree ($\emptyset$) and flip its quantifiers ($\neg$). On the right side, the argument is applied to `add_vote(v, c, s)`, which itself has a non-empty tree. Therefore, we apply α to $\mathcal{F}$`Votes`.$\emptyset$ in that case.

Abstract quantifier interaction trees provide useful information to the developer about what arguments they can supply to functions. For example, the abstract tree for `is_invariant` tells us that we can't apply `in_invariant` to a function that existentially quantifies a `Votes`—that would create an $\exists$`Votes`. $\{\mathcal{F}\texttt{Votes}.\ldots\}$ alternation, which would in-turn create a cycle in the sort graph.

2.4 Higher-Order Functions

So far, we have illustrated how our verification condition encoding addresses the challenge of decidability. However, the presence of higher-order functions in our language presents another obstacle: Extended EPR is a fragment of *first*-order logic, which does not allow higher-order functions.

In our Fig. 1 example, we can eliminate the higher-order `is_invariant` function by partial evaluation before encoding, but the `filter` function is imported, and must remain as an uninterpreted symbol. A further problem is that the axiom for `filter` seems to use a higher-order quantifier:

```
#[assume(filter(f,s1) => s2)] fn filter_def() -> bool {
    forall(|p1: Pid| member(p1,s2) == (member(p1,s1 && f(p1)))) }
```

The `#[assume(..)]` syntax means that the body of the `fn` is assumed for any `f`, `s1`, and `s2` such that `filter(f,s1)` produces `s2`. In general, because we do not allow higher-order quantifiers except at the top-level of axioms, functions can only be applied to a finite set of concrete higher-order arguments within the scope of any particular assertion. The higher-order quantifier in the `filter` axiom is similarly eliminated when we replace the axiom with a `filter1` axiom that has "hard-coded" the function argument:

```
#[assume(filter1(s1) => s2)] fn filter_def1() -> bool {
    let f = |(v,_)| v == p
    forall(|p1: Pid| member(p1,s2) == (member(p1,s1) && f(p1))) }
```

Note that the axiom has now captured the `p` value from the body of `has_voted` where `filter` was called. To address this, we instantiate the `filter1` axiom in-place, rather than keeping it as a universally-quantified top-level axiom. This means that the `∀Pid` quantifier in the axiom is added to the assertion, which affects its quantifier interaction tree. Like the $\mathcal{F}$ quantifier node itself, quantifiers under an $\mathcal{F}$—which only come from instantiated axioms—are protected from flipping due to negations higher-up in the tree.

2.5 Inductive Datatypes and Recursive Functions

We've shown how to use assumptions on imported, possibly-complex functions to verify assertions on our own, relatively simple code. From the solver's point of view, imported functions are *uninterpreted* while our own code is *interpreted*.

This changes when our own code becomes complex by performing recursion. As an example, consider the following recursive function that pattern-matches on `enum Nat { Z, S(Nat) }`, an inductively defined type:

```
fn add(n1: Nat, n2: Nat) -> Nat { match n1 {
    Z => n2,
    S(n1_minus) => S(add(n1_minus,n2)),
}}
```

Now let's verify a property of `add`, assuming `<=` as an imported `Nat` relation:

```
#[verify] fn add_prop1() -> bool {
    forall(|x: Nat, y: Nat| (x <= y) => (add(x,0) <= add(y,0))) }
```

This assertion fails to verify. When our code calls `add`, we can't simply replace the call with `add`'s body—the body itself contains `add` again. In fact, the solver leaves `add` uninterpreted, like an imported function, and assumes nothing except functionality about its output. Thus, to verify a property like `add_prop` we'll need to give `add` an axiom—not a *trusted* axiom, like on an imported function, but a *checked* axiom, which we'll call an *annotation*.

```
#[annotate(add(n1,Z) => n3)] fn add_zero_right() -> bool { n1 == n3 }
```

This annotation says that, when `Z` is the second argument to `add`, then the output (`n3`) is the same as the first argument (`n1`). To verify this, Ravencheck assumes the annotation, as an inductive hypothesis, and then attempts to verify the annotation with `add`'s body unrolled by one step.

```
n1 == match n1 { Z => Z, S(n1_minus) => S(add(n1_minus,Z)) }
```

When `n1` is `Z`, clearly `Z = Z`, and when `n1` is `S(n1_minus)`, the hypothesis yields `add(n1_minus,Z) = n1_minus`, and so verification succeeds. Once this annotation has been checked and assumed for all calls to `add`, verification for `add_prop1` also succeeds.

For termination, we require recursive call arguments to be related to the initial call arguments by a well-founded relation—in this case, we use the relation between an inductive data value (`n`) and its sub-value (`n_minus`).

2.6 Quantifier Instantiation

Our partial function semantics for Λ_{EPR} is motivated by the incompleteness caused by relational abstraction. We can use the partial function semantics to diagnose or anticipate this incompleteness, but how do we fix it? For example, consider the following, failing verification condition on the inductive `Nat` type:

$$
\begin{array}{llll}
Q & ::= & \exists D.\, W \mid \forall D.\, W \mid \mathcal{F} D.\, W \mid \alpha \mid \neg\alpha & \textit{quant. inter. node} \\
W & ::= & \{Q, \ldots, Q\} \mid \Lambda\alpha.\, W \mid \alpha(W) & \textit{quant. inter. tree} \\
T & ::= & \mathbf{Data}(D) \mid \mathbf{Prop} \mid T \to T & \textit{simple type}
\end{array}
$$

Fig. 2. Grammar of types. The full type of a term takes the form $\langle T, W \rangle$, containing the simple type T and the quantifier interaction tree W.

```
#[verify] factor_s_left() -> bool {
    forall(|a: Nat, b: Nat| add(S(a), b) == S(add(a,b))) }
```

Intuitively, two assumptions on `Nat` should suffice: commutativity (`add(a,b) == add(b,a)`), and a property allowing `S` to factor out of the right argument (`add(a, S(b)) == S(add(a,b))`. If we were writing a manual proof, this would allow us to transform the goal into an identity:

```
    add(S(a), b) == S(add(a, b))) ==> add(b, S(a)) == S(add(a, b)))
==> S(add(b, a)) == S(add(a, b))) ==> S(add(a, b)) == S(add(a, b)))
```

However, even if we provide these assumptions, the solver cannot verify the condition. This is because two of the intermediate values—the specific calls `add(b, S(a))` and `S(add(b, a))`—are not *instantiated*. They do not appear in the verification condition, and so the partial semantics allows them to be *undefined*, in which case the commutativity and factor-right assumptions for those values are ignored.

We can fix this by simply adding those values to the verification condition:

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

This technique is called *quantifier instantiation*, and has been explored in existing work [Tamir et al. 2023]. However, we consider our solution using let-bindings to significantly simplify the user interface for applying this technique.

3 Syntax and Decidability

In this section, we present the abstract syntax and type system of Λ_{EPR}. The syntax combines purely functional programming with quantified logical expressions, and the type system tracks quantifier alternations in order to determine the decidability of assertions.

In Sec. 2, *imported functions*, provided by a library, are a key facet of our examples and verification solution. In this and the following sections, we formalize imports abstractly as *primitive operators*, with trusted axioms, without detailing a mechanism for packaging and importing them. In practice, those axioms would be verified in the context of the packages that export them.

3.1 Quantifier Interaction Types

Beyond the typical execution guarantees provided by type-safety, users of Λ_{EPR} are concerned with a guarantee for their verification assertions: decidability. This property depends on several aspects of user code and their libraries' interfaces: the nesting of quantifiers in user assertions, the use of imported functions (i.e. primitive operators) in proximity to quantifiers, and the use of quantifiers in the axioms for those imported functions.

To mitigate this complexity, we track the interaction between quantifiers and operators in the type system of Λ_{EPR}.

Types are divided into a *simple* part and a *quantifier interaction tree* part, as described in Fig. 2. *Proposition* terms have the simple type **Prop**, and denote boolean values. *Data object* terms have simple type $\mathbf{Data}(D)$ for some *data sort* D, and denote data of that sort. *Function* terms have simple type $T_1 \to T_2$. Simple types are paired with a *quantifier interaction tree* W, which specifies the nesting of quantifiers (and primitive operator applications) contained within the denoted term.

For example, consider the following first-order logic assertion:

$$(\forall x : D_1.\ \exists y : D_2.\ \varphi(x, y)) \wedge (\forall x : D_3.\ \forall y : D_3.\ f(x, y) = f(y, x).$$

Here, φ is a predicate of type $\mathcal{P}(D_1 \times D_2)$, and f is a function of type $D_3 \times D_3 \to D_3$. This assertion can be represented by a Λ_{EPR} term with type $\langle \mathbf{Prop}, W \rangle$, where W takes the form:

$$\{\forall D_1.\ \{\exists D_2.\emptyset\},\ \forall D_3.\ \{\forall D_3.\ \{\mathcal{F} D_3.\emptyset\}\}\}.$$

Because both uses of f produce D_3 values, the tree has $\mathcal{F} D_3.\emptyset$ as a leaf beneath the two $\forall D_3$ nodes. The $\emptyset$ underneath indicates that all axioms on f are quantifier-free.

Executability. A Λ_{EPR} term is *executable* if it does not contain quantifiers. The type $\langle T, W \rangle$ denotes this executability property if W takes the form $\{\mathcal{F} D_1.\ W_1, \ldots, \mathcal{F} D_n.\ W_n\}$. The subtrees $W_1, \ldots, W_n$ describe quantifiers in the axioms of the operators that are used, which are not relevant to executability.

Decidability. A Λ_{EPR} term is a *decidable assertion* if it can be encoded into the Extended EPR fragment of logic. A term with type $\langle \mathbf{Prop}, W \rangle$ is a decidable assertion if W does not generate a *sort graph* that contains cycles.

Definition 3.1 (Sort graph). The *sort graph* of a quantifier interaction tree W, written as $\mathbf{Graph}(W)$, is a graph with data sorts as nodes. The graph contains an edge $D_1 \to D_2$ iff W contains a subtree $\exists D_1.W_1$ and W_1 contains either $\forall D_2.W_2$ or $\mathcal{F} D_2.W_2$. This graph is undefined for *abstract* trees of the form $\Lambda\alpha.W$.

Intuitively, the quantifier alternation $\exists D_1.\forall D_2 \ldots$ creates an edge, and an application of a primitive operator with output type D is treated like a $\forall D$ quantification.

Definition 3.2 (Decidable tree). We say that W is *decidable* iff $\mathbf{Graph}(W)$ is acyclic.

The easiest way to break decidability is to use alternating quantifiers on the same sort, like $\exists D.\forall D.\varphi$. However, combinations of alternations on different sorts can also break decidability, like:

$$(\exists D_1.\forall D_2.\varphi_a) \quad \wedge \quad (\exists D_2.\forall D_1.\varphi_b.)$$

For this reason, *composition* of Λ_{EPR} terms is relevant to decidability—we cannot determine decidability by only looking at each component part separately.

Abstraction. Functions in Λ_{EPR} have abstracted trees, of the form $\Lambda\alpha.W$. The decidability of a function depends on the argument given to it, and likewise its quantifier interaction tree is defined in terms of the argument's tree, which is substituted in for α.

Negation. The *negation* of a quantifier interaction tree W, written as $\neg W$, is a tree with the same shape, but with $\forall$ and $\exists$ quantifiers flipped. $\mathcal{F}$ nodes are not changed, and protect their subtrees from changing as well. Negation cannot be applied to Λ-abstraction trees.

$$\begin{aligned}
\neg\,\{Q_1, \ldots, Q_n\} &\triangleq \{\neg\, Q_1, \ldots, \neg\, Q_n\} \\
\neg\,\forall D.\, W &\triangleq \exists D.\, \neg W \\
\neg\,\exists D.\, W &\triangleq \forall D.\, \neg W \\
\neg\,\mathcal{F} D.\, W &\triangleq \mathcal{F} D.\, W
\end{aligned}$$

$$
\begin{array}{lll}
B & ::= & \mathbf{and} \mid \mathbf{or} \qquad \textit{connectives} \\
E & ::= & x \mid \mathbf{fun}[T](x.\ E) \mid \mathbf{apply}(E, E) \qquad \textit{expressions} \\
 & & \mid \mathbf{let}(E,\ x.\ E) \mid \mathbf{eval}(E,\ x.\ E) \mid \mathbf{ite}(E,\ E,\ E) \mid \mathbf{match}(E, \{(S(x, \ldots), E), \ldots\}) \\
 & & \mid \mathbf{const}[S] \mid \mathbf{op}[S](E, \ldots, E) \mid \mathbf{pred}[S](E, \ldots, E) \\
 & & \mid \mathbf{eq}[D](E, E) \mid \mathbf{connect}[B](E, E) \mid \mathbf{true} \mid \mathbf{false} \mid \mathbf{not}(E) \\
 & & \mid \mathbf{forall}[D](x.\ E) \mid \mathbf{exists}[D](x.\ E)
\end{array}
$$

Fig. 3. Grammar of expressions.

$$\boxed{\Delta; \Gamma \vdash E : T, W}$$

$$\frac{\Gamma(x) = \langle T, W \rangle}{\Delta; \Gamma \vdash x : T, W} \qquad \frac{\begin{array}{c}\mathbf{TypeOf}(S) = T_1 \to \cdots \to T_n \to \mathbf{Data}(D) \qquad W = \Delta(S)[W_1/\alpha_1] \cdots [W_n/\alpha_n] \\ \Delta; \Gamma \vdash E_1 : T_1, W_1 \qquad \cdots \qquad \Delta; \Gamma \vdash E_n : T_n, W_n\end{array}}{\Delta; \Gamma \vdash \mathbf{op}[S](E_1, \ldots, E_n) : \mathbf{Data}(D),\ (\mathcal{F} D.\ W) \cup W_1 \cup \cdots \cup W_n}$$

$$\frac{\Delta; \Gamma \vdash E : \mathbf{Prop}, W}{\Delta; \Gamma \vdash \mathbf{not}(E) : \mathbf{Prop}, \neg W} \qquad \frac{\Delta; \Gamma, x : \langle \mathbf{Data}(D), \emptyset \rangle \vdash E : \mathbf{Prop}, W}{\Delta; \Gamma \vdash \mathbf{forall}[D](x.\ E) : \mathbf{Prop}, \{\forall D.\ W\}}$$

$$\frac{\Delta; \Gamma, x : \langle \mathbf{Data}(D), \emptyset \rangle \vdash E : \mathbf{Prop}, W}{\Delta; \Gamma \vdash \mathbf{exists}[D](x.\ E) : \mathbf{Prop}, \{\exists D.\ W\}} \qquad \frac{\Delta; \Gamma, x : \langle T_1, \alpha \rangle \vdash E : T_2, W}{\Delta; \Gamma \vdash \mathbf{fun}[T_1](x.\ E) : T_1 \to T_2, \Lambda\alpha.W}$$

$$\frac{\begin{array}{c}\Delta; \Gamma \vdash E_1 : T_1 \to T_2, \Lambda\alpha.W \\ \Delta; \Gamma \vdash E_2 : T_1, W_1\end{array}}{\Delta; \Gamma \vdash \mathbf{apply}(E_1, E_2) : T_2, W[W_1/\alpha]} \qquad \frac{\begin{array}{c}\Delta; \Gamma \vdash E_1 : T_1, W_1 \\ \Delta; \Gamma, x : \langle T_1, W_1 \rangle \vdash E_2 : T_2, W_2\end{array}}{\Delta; \Gamma \vdash \mathbf{let}(E_1,\ x.\ E_2) : T_2, W_2}$$

$$\frac{\Delta; \Gamma \vdash E_1 : \mathbf{Data}(D), W_1 \qquad \Delta; \Gamma, x : \langle \mathbf{Data}(D), \emptyset \rangle \vdash E_2 : T, W_2}{\Delta; \Gamma \vdash \mathbf{eval}(E_1,\ x.\ E_2) : T, W_1 \cup W_2}$$

Fig. 4. Type rules for expressions. The module context Δ carries quantifier interaction information about axioms defined at the module level, outside of any expression, which is relevant to the **op** term. We elide several trivial cases—the full rules can be found in the appendix.

Intuitively, $\mathcal{F}$ nodes are unaffected by negation because, as we will describe in Sec.5, assertions are placed in negation-normal-form before function calls are replaced by quantifiers. This is safe because these introduced quantifiers each represent one-and-only-one value (the output of the abstracted function), and so it is sound to use either universal or existential quantification.

Note that decidability-checking is conservative. For example, the formula $\exists a : A, b : B.\ r(a, f(b))$, where $f : B \to A$, has quantifier tree $\{\exists A.\{\exists B.\{FA.\emptyset\}\}\}$, generating an $A \to A$ edge even though the formula is in Extended EPR. The **eval** term (Sec. 3.2) can be used to rewrite this example without changing its semantics, so that its quantifier tree is acyclic: $\exists b : B.\ \mathbf{eval}\ a2 := f(b).\ \exists a : A.r(a, a2)$. In our experience with case studies, we have not needed to perform any such rewrites.

3.2 Expressions

The grammar of expressions is given in Fig. 3, and the type rules are given in Fig. 4.

Primitive Operators. A primitive operator is identified by a symbol S. We assume operators have static simple types assigned by $\mathbf{TypeOf}(S)$. Three kinds of operators are used in Λ_{EPR}: *predicate* operators that have a **Prop** output type, *function* operators that have a $\mathbf{Data}(D)$ output type, and *constant* operators have a data output and no input arguments. Operators cannot return functions as output. Function operators are assigned abstract quantifier interaction trees $\Delta(S)$ by the *module context* Δ.

The $\mathbf{pred}[S](E_1, \ldots, E_n)$ term applies a predicate operator to arguments of the correct type. The resulting quantifier alternation tree is the union of the arguments' trees. The $\mathbf{op}[S](E_1, \ldots, E_n)$ term is similar but for function operators—unlike predicate operators, this term can take higher-order arguments. Its quantifier alternation tree is the union of its arguments' trees, combined with the application of those trees to the operator's abstract tree $\Delta(S)$.

Let and Eval. A notable distinction must be made between **let** and **eval**. The Λ_{EPR} language is free of side-effects, including non-termination, and so evaluation order does not impact semantics. However, the *encoding order*—the order in which relational abstraction quantifiers are nested—can impact *decidability*. Therefore, we provide the **eval** binding for data-type terms, which "eagerly" places their quantifiers in contrast to the "lazy" **let** binding. The difference can be seen in the types: for **eval**, the bound term's tree W_1 is placed alongside the body's tree W_2, while for **let** the bound term's tree enters the context, so that it appears as a subtree of W_2 or not at all. At a meta level, we will use **eval** in order to define a normal form for expressions that simplifies our verification encoding. Users of Λ_{EPR} can use also **eval** to fix decidability in some cases.

3.3 Normal Form

To simplify reasoning about Λ_{EPR} expressions, and to prepare them for verification encoding, we transform them into a normal form. First, partial evaluation is performed, for which we elide the detailed formalization. This evaluation β-reduces **fun** and **apply** terms, and places assertions in *negation-normal form*. After partial evaluation, we *unroll* **op** terms so that expressions take on a form similar to the *call-by-push-value* λ-calculus [Levy 1999]. Intuitively, this unrolling looks like:

$$\exists a, b.\ \neg p(f(a), g(b))) \qquad \rightarrow \qquad \exists a, b.\ \mathbf{let}\, x_1 = f(a);\ \mathbf{let}\, x_2 = g(b);\ \neg p(x_1, x_2)$$

This prepares us for the relational-abstraction encoding, which replaces the outputs of functions with quantified variables (where f^r is the relation representing f):

$$\rightarrow \qquad \exists a, b.\ \forall x_1 . f^r(a, x_1) \implies \forall x_2 . g^r(b, x_2) \implies \neg p(x_1, x_2)$$

For a partially-evaluated expression E, this unrolling is performed by the functions $\mathbf{Eval}(\mathbf{Unroll}(E))$, which we fully define in the appendix. Applying $\mathbf{Unroll}(E)$ produces a pair $\langle \Pi, E' \rangle$, where E' has had its $\mathbf{op}[S](\ldots)$ terms removed and replaced by variables, and Π specifies how to assign those variables.

$$\Pi \quad ::= \quad \epsilon \mid \Pi, S(E, \ldots, E \rightsquigarrow x)$$

Each $S(E_1, \ldots, E_n \rightsquigarrow x)$ in Π indicates that x should be the output of a term $\mathbf{op}[S](E_1, \ldots, E_n)$. Intuitively, when **op** terms are nested, this process turns them inside-out: the inner applications are evaluated (by **eval** terms) before the outer applications.

To demonstrate, consider the following application:

$$\mathbf{Unroll}(\mathbf{pred}[S_p](\mathbf{op}[S_1](y, \mathbf{op}[S_2](f), \mathbf{op}[S_3](z)))) \ = \\ \langle S_2(f \rightsquigarrow x_2), S_3(z \rightsquigarrow x_3), S_1(y, x_2, x_3 \rightsquigarrow x_1) \rangle\,,\ \mathbf{pred}[S_p](x_1).$$

When we apply **Eval** to this pair, it adds a prefix of **eval** terms to $\mathbf{pred}[S_p](x_1)$:

$$\mathbf{eval}(\mathbf{op}[S_2](f),\ x_2.\ \mathbf{eval}(\mathbf{op}[S_3](z),\ x_3.\ \mathbf{eval}(\mathbf{op}[S_1](y, x_2, x_3),\ x_1.\ \mathbf{pred}[S_p](x_1))))$$

This prepares us to replace each of the **eval** terms with a universal quantifier, guarded by the relational abstraction (S^e of each operation S):

$$\forall x_2.\ (S_2^e(f, x_2)) \implies \cdots \implies \forall x_1.\ (S_1^e(y, x_2, x_3)) \implies \mathbf{pred}[S_p](x_1)$$

We describe this relational-abstraction encoding step in more detail in Sec. 5.

Theorem 3.3 (Normal Form Preservation). *The transformation of a* Λ_{EPR} *expression into normal form does not alter its type, including the quantifier tree.*

3.4 Modules

A complete verification problem, expressed in Λ_{EPR}, consists of a set of axioms (of various forms: `assume`, `assume for`, `annotate for`), and a set of assertions. We formalize this as a *verification module* L, taking the form:

$$\left\langle E_1^a, \ldots, E_j^a;\ F_1, \ldots, F_k;\ C_1, \ldots, C_l\ E_1^s, \ldots, E_m^s \right\rangle$$

Here, each F is a function-axiom of the form $\langle S;\ x_1, \ldots, x_n;\ x;\ E\rangle$, corresponding to the `#[assume(S(x1,...,xn))] fn my_axiom() -> { E }` syntax from our examples in Sec. 2. Likewise, each C is a combination of recursive function definition and annotation, of the form $\langle S;\ x_1, \ldots, x_n;\ E_d\ x;\ E_a\rangle$. Here, E_d is the body of the recursive function definition, which can refer to $x_1, \ldots, x_n$, and E_a is the body of the annotation, which can refer additionally to x. Each E^a is an assumption, and each E^s is an assertion (i.e. goal).

Intuitively, a module is valid when, given the assumptions, (1) each recursive definition satisfies its annotation, and (2) each assertion is valid. To determine the decidability of this verification problem, we combine the quantifier interaction trees of all the module's components, and check its sort graph.

Definition 3.4 (Decidable module). A verification module

$$\left\langle E_1^a, \ldots, E_j^a;\ F_1, \ldots, F_k;\ C_1, \ldots, C_l;\ E_1^s, \ldots, E_m^s \right\rangle$$

is *decidable* under the following conditions. Let Δ be the *module context*. For each primitive functional operator S, we assign $\Delta(S)$ to be the union of each $\neg\, W$ for which an $F_i = \langle S;\ x_1, \ldots, x_n;\ x;\ E\rangle$ and $E : W$. For each recursively defined operator S, we assign $\Delta(S)$ to be the union of each $\neg W$ for which a $C_i = \langle S;\ x_1, \ldots, x_n;\ E_d\ x;\ E_a\rangle$ and $E_a : W$. Let W_a be the *assumption tree*: a union of each $\neg\, W_i$ for which $E_i^a : W_i$. Now, the following trees must be decidable.

(1) $\neg\, W_a \cup W_i$ for each $\Delta, \epsilon \vdash E_i^s : W_i$.
(2) $\neg\, W_a \cup W_i$ for each $C_i = \langle S;\ x_1, \ldots, x_n;\ E_d\ x;\ E_a\rangle$ such that $\Delta, \epsilon \vdash E_a[E_d/x] : W_i$.

Note that the trees of function axioms and recursive function annotations are not negated at the top level like those of the simple axioms; rather, they go into the typing context. This is because they will be instantiated in-place in our verification encoding.

4 Semantics as HOL

We consider the semantics of our language over two different domains: that of *partial functions*, and that of *total functions*. We require that all primitive operators be total, and so the semantics of program executions is restricted to the domain of total functions. However, our verification technique requires us to encode conditions into the Extended EPR fragment of FOL, in which we cannot express our assumption of totality for all primitive functions. Therefore, we define the

semantics of verification over the domain of partial functions. The partial functions are a superset of the total functions, and so the result of this arrangement is that verification is *sound*, but *not complete*, with respect to the execution semantics.

In Sec 5, we will show that the partial-function semantics is sound with respect to our actual verification encoding, and also complete when higher-order primitive operators are omitted.

4.1 HOL over Partial Functions

Our execution and verification semantics are defined by interpreting Λ_{EPR} expressions into *higher-order logic*—specifically *simple type theory* (STT) [Farmer 2008]. We use higher-order logic in order to represent Λ_{EPR}'s support for higher-order functions. Our language does not support higher-order quantifiers.

Syntax. We extend the standard syntax of STT with a term that mirrors the **eval** term from Λ_{EPR}'s abstract syntax. The grammar of our extended STT is as follows.

$$A \quad ::= \quad x \mid S \mid A(A) \mid \lambda x.\, A \mid A = A \mid A \rightsquigarrow x.\, A$$

Semantics. A *signature* is a pair $\langle \mathtt{Symbol}, \mathtt{SType} \rangle$, where $\mathtt{Symbol}$ is the set of constant symbols, and $\mathtt{SType}$ maps each member of $\mathtt{Symbol}$ to a type in set $\mathbf{Types}(\mathcal{D})$. The set $\mathbf{Types}(\mathcal{D})$ is defined inductively: (1) $\mathbf{Prop} \in \mathbf{Types}(\mathcal{D})$, (2) for every domain $D \in \mathcal{D}$, we have $\mathbf{Data}(D) \in \mathbf{Types}(\mathcal{D})$, and (3) for every $T_1, T_2 \in \mathbf{Types}(\mathcal{D})$, we have $T_1 \to T_2 \in \mathbf{Types}(\mathcal{D})$.

A *partial model* for a signature $\langle \mathtt{Symbol}, \mathtt{SType} \rangle$ is a pair $M = \langle \mathcal{U}, I \rangle$, where:

(1) $\mathcal{U} = \{\, U_T \mid T \in \mathbf{Types}(\mathcal{D}) \,\}$ is a set of non-empty domains, one for each type in $\mathbf{Types}(\mathcal{D})$.
(2) $U_{\mathbf{Prop}}$, the domain of truth values, is the set $\{\mathtt{True}, \mathtt{False}\}$.
(3) For $T_1, T_2 \in \mathbf{Types}(\mathcal{D})$, $U_{T_1 \to T_2}$ is the set of partial functions from U_{T_1} to U_{T_2}.
(4) I maps each $S \in \mathtt{Symbol}$ to a member of $U_{\mathtt{SType}(S)}$. If $\mathtt{SType}(S) = T \to \mathbf{Prop}$, then $I(S)$ must be a total function.

Our semantics is based on a particular interpretation of $=$ for partial functions. If f_1 and f_2 both define outputs for some input V, and those inputs differ, then $f_1 \neq f_2$. If f_1 and f_2 are total, and their outputs match for all inputs, then $f_1 = f_2$. Otherwise, $f_1 = f_2$ is undefined.

An expression A has a *valuation* in model M under variable assignment Σ, written $\mathbf{Value}^M_\Sigma A$, which may be undefined. We define the valuation as follows.

- Let A be the variable x. Then $\mathbf{Value}^M_\Sigma A = \Sigma(x)$.
- Let A be the constant symbol S. Then $\mathbf{Value}^M_\Sigma A = I^M(S)$.
- Let A be $A_1(A_2)$. If $\mathbf{Value}^M_\Sigma A_1$ and $\mathbf{Value}^M_\Sigma A_2$ are both defined, and the function $\mathbf{Value}^M_\Sigma A_1$ defines output for input $\mathbf{Value}^M_\Sigma A_2$, then $\mathbf{Value}^M_\Sigma A$ is that output. Otherwise, $\mathbf{Value}^M_\Sigma A$ is undefined.
- Let A be $A_1 \rightsquigarrow x.\, A_2$. If $\mathbf{Value}^M_\Sigma A_1$ is undefined, then $\mathbf{Value}^M_\Sigma A$ is also undefined. Otherwise, let V be $\mathbf{Value}^M_\Sigma A_1$, and $\mathbf{Value}^M_\Sigma A = \mathbf{Value}^M_{\Sigma[x \mapsto V]} A_2$.
- Let A be $\lambda x : T.\, A_1$. Then $\mathbf{Value}^M_\Sigma A$ is the function f such that, for each $V \in U_T$, $f(V) = \mathbf{Value}^M_{\Sigma[x \mapsto V]} A_1$.
- Let A be $A_1 = A_2$. If $\mathbf{Value}^M_\Sigma A_1 = \mathbf{Value}^M_\Sigma A_2$ has a defined truth value, then $\mathbf{Value}^M_\Sigma A$ is that value. Otherwise, $\mathbf{Value}^M_\Sigma A$ is undefined.

We perform the standard translation [Farmer 2008] of logical operators into our variant of STT. For example:

$$\forall x : T.\, A \quad \triangleq \quad (\lambda x : T.\, A) = (\lambda x : T.\, \mathtt{True}).$$

∧	**T**	**U**	**F**
T	T	U	F
U	U	U	F
F	F	F	F

∨	**T**	**U**	**F**
T	T	T	T
U	T	U	U
F	T	U	F

¬	**T**	**U**	**F**
	F	U	T

Fig. 5. Truth tables for logical operators in STT with partial-function semantics. The U values stand for *undefined*.

When our particular definition of = on partial functions is considered, $\forall x : T. A$ takes on the following behavior:

- When any single choice of x makes A false, the whole $\forall$ expression is false (because the two functions mismatch on a defined output).
- When A is defined as true for every choice of x, the whole expression is true (because $\lambda x : T.\,\texttt{True}$ is total as well).
- Otherwise, when choices of x can either make A true or undefined, the whole expression is undefined.

Similarly, we summarize the behavior of ¬,∧, and ∨ in Fig. 5. This behavior matches that of an existing *three-value logic* called the *logic of paradox* [Arieli et al. 2011; Priest 1979]. This logic is *paraconsistent* in the sense that $A \wedge \neg A$ is not necessarily false: if A is undefined, then $A \wedge \neg A$ is also undefined.

We say that an STT expression A is *valid*, written as $\models A$, iff its valuation is `True` or undefined for every model. Because undefinedness arises solely from non-total functions, paraconsistency does not extend to validity. Validity considers all models in which every function is total, in addition to those with non-total functions, and every total-function-only model assigns $A \wedge \neg A$ to `False`. Thus, the contradiction may not be false in every model, but it is always *invalid*.

If we only consider models that assign all symbols to *total* functions, our semantics behaves exactly like standard STT semantics: no expression can be undefined. Therefore, our special semantics for STT is sound (but not complete) with respect to standard semantics.

Theorem 4.1 (Partial implies total). *If an STT expression is valid under partial-function semantics, then it is also valid under total-function semantics.*

4.2 Expression and Module Semantics

We interpret Λ_{EPR} expressions in our language as formulas in STT[2], and define a Λ_{EPR} module (Sec. 3.4) as valid if all assertions pass (`True` or undefined) for every model that satisfies the function axioms and recursive function annotations.

We cannot simply have function axioms imply the assertions, as we can with the regular axioms, because that would only check the assertions on total models—we must also consider the partial-function models, in order to match the behavior of the verifier. For example, consider the function axiom $A \triangleq \forall \texttt{insert}(e, s) \rightsquigarrow s_2.\ \texttt{member}(e, s_2)$, and the assertion $P \triangleq \forall e\ \exists s.member(e, s)$. The naive encoding $(\forall e, s.\ \texttt{member}(e, \texttt{insert}(e, s))) \implies P$ is valid, even though verification would actually fail. Every model that falsifies P by not containing a satisfying s evaluates the implication to `Undefined`, because the axiom is `Undefined` in that case, and `Undefined` ∨ `False` is `Undefined`. Only a model that totally defines `insert` would allow the implication to evaluate to `False`, and in that total case, P must be `True`.

[2]The full translation is given in the appendix

Definition 4.2 (Module semantics). A verification module

$$\left\langle E^a_1, \ldots, E^a_j;\ F_1, \ldots, F_k;\ C_1, \ldots, C_l\ E^s_1, \ldots, E^s_m \right\rangle$$

is *valid* under the following conditions. Let $\mathcal{M}$ be the set of models (including partial models) that satisfy the following.

(1) For each $M \in \mathcal{M}$, and each $F_i = \langle S;\ x_1, \ldots, x_n;\ x;\ E\rangle$ such that $\texttt{SType}(S) = T_1 \to \cdots \to T_n \to T$, and each $V_1 \in U_{T_1}, \ldots, V_n \in U_{T_n}, V \in U_T$, if $I^M(S)(V_1)\cdots(V_n) = V$, then $\mathbf{Value}^M_{[x_1 \mapsto V_1, \ldots x_n \mapsto V_n, x \mapsto V]}[\![E]\!]^{\mathsf{h}} = \texttt{True}$.
(2) For each $M \in \mathcal{M}$, and each $C_i = \langle S;\ x_1, \ldots, x_n;\ E_d\ x;\ E_a\rangle$ such that $\texttt{SType}(S) = T_1 \to \cdots \to T_n \to T$, and each $V_1 \in U_{T_1}, \ldots, V_n \in U_{T_n}, V \in U_T$, if $I^M(S)(V_1)\cdots(V_n) = V$, then $\mathbf{Value}^M_{[x_1 \mapsto V_1, \ldots x_n \mapsto V_n, x \mapsto V]}[\![E_a]\!]^{\mathsf{h}} = \texttt{True}$.

Let A_a be the conjunction of each $[\![E^a_i]\!]^{\mathsf{h}}$. Now, the module is valid if, for every model $M \in \mathcal{M}$, the following conditions hold.

(1) For each E^s_i, $\mathbf{Value}^M(A_a \implies [\![E^s_i]\!]^{\mathsf{h}})$ is $\texttt{True}$ or undefined.
(2) For each $C_i = \langle S;\ x_1, \ldots, x_n;\ E_d\ x;\ E_a\rangle$, $\mathbf{Value}^M(A_a \implies \forall x_1, \ldots, x_n.\ [\![E_a[E_d/x]]\!]^{\mathsf{h}})$ is $\texttt{True}$ or undefined.

Example 4.3. Suppose we have a primitive function $\texttt{increment} : \texttt{Nat} \to \texttt{Nat}$. We cannot directly apply $\texttt{increment}$ within an axiom, since that would create a $\texttt{Nat} \leftrightarrow \texttt{Nat}$ sort-cycle, and so we use a function axiom as follows:

$$\forall(\texttt{increment}\ x_1 \rightsquigarrow x_2).\quad x_1 < x_2.$$

Now consider two assertions:

$$\begin{array}{lcl} P_1 & \triangleq & \forall y : \texttt{Nat}.\ \exists z : \texttt{Nat}.\quad y < z. \\ P_2 & \triangleq & \forall x : \texttt{Nat}.\quad x < \texttt{increment}(x) \end{array}$$

The validity of P_1 is straightforward: in a model that defines $\texttt{increment}$ as a total function, the output of $\texttt{increment}(y)$ is a satisfying choice for z. In a model that defines $\texttt{increment}$ as partial, there is some y which has $\texttt{increment}(y) = \bot$. For that y, it is possible the model contains no satisfying z. Thus, P_1 is valid for total models, but not for partial models.

The validity of P_2 invokes our three-valued semantics. In a model that defines $\texttt{increment}$ as a total function, P_2 is clearly $\texttt{true}$ based on the axiom. But what about a model with partial $\texttt{increment}$? In that case, P_2 is undefined: for each x that has an increment, the $\forall$ body holds due to the axiom, and for each x that has an undefined increment, the body is undefined. If $\forall$ gets a mix of true and undefined cases, its overall value is undefined.

To consider $P_1 \wedge P_2$ together, we can consult the truth table in Fig. 5. For a model with total $\texttt{increment}$, both clauses are true. But for a model with partial $\texttt{increment}$, we have $\texttt{False} \wedge \texttt{Undefined}$. In that case, the overall value is $\texttt{False}$. In practice, this means that $P_1 \wedge P_2$ would always hold for real executions, where $\texttt{increment}$ is implemented by a total function, but it will not hold under verification, where decidability does not allow us to assume totality. And thus verification is sound, but not complete.

5 Encoding into FOL

Our automated verification approach sends assertion expressions to an SMT solver. This requires us to reduce expressions to first-order logic, without any higher-order functions or higher-order quantifiers. Furthermore, our encoding performs relational abstraction on all primitive function operators, making the conditions more amenable to the restrictions of the Extended EPR fragment.

5.1 Encoding

We encode expressions of the form $E = (E_{\texttt{axioms}} \implies E_{\texttt{assertion}})$, for some assertion (with simple type **Prop**) in a module. The $E_{\texttt{axioms}}$ expression is the conjunction of all ordinary axioms (not function axioms or recursive function annotations) in the module. The function axioms and recursive function annotations will be instantiated as necessary during the encoding process.

Our first step is to normalize E according to the procedure described in Sec. 3.3. We will now define a function $[\![\cdot]\!]^{\mathrm{e}}$ that performs the encoding on a normalized assertion.

After normalization, **op** terms only appear in the form

$$\mathbf{eval}(\mathbf{op}[S](E_1, \ldots, E_n),\ x.\ E).$$

If S is first-order, $E_1, \ldots, E_n$ are each either a variable (x_1) or a constant $\mathbf{const}[S]$. If S is higher-order, then any argument could be a function with an arbitrarily-complex body. Now, suppose that the module defined a function axiom for S: $\langle S;\ x_1, \ldots, x_n;\ x;\ E_a \rangle$. Whatever the arguments are, we substitute them into the function axiom body, normalize the result, and then encode it as well:

$$\forall x.\ (S^e([\![E_1]\!]^{\mathrm{e}}, \ldots, [\![E_n]\!]^{\mathrm{e}}) \wedge [\![\mathbf{Norm}(E_a[E_1/x_1, \ldots, E_n/x_n])]\!]^{\mathrm{e}}) \implies [\![E]\!]^{\mathrm{e}}.$$

Note that, in addition to instantiating the axiom, we relate the inputs to the output using the relation symbol S^e. This allows us to assume a functionality axiom for S^e, allowing us to assert that $\mathbf{op}[S](\ldots)$ always gives the same output for the same input. Inductive datatype constructors receive an additional axiom: two constructors that produce equal output values must themselves be equal. Unfortunately, higher-order arguments cannot be related by the first-order symbol S^e, and so equality comparison on the outputs of higher-order functions is incomplete.

We use the same rules for recursively-defined function operators that have annotation axioms. For if-then-else terms, we assert the condition's truth on the **then** branch and the condition's falsity on the **else** branch.

$$\begin{aligned}&[\![\mathbf{if}\ E_1\ \mathbf{then}\ E_2\ \mathbf{else}\ E_3]\!]^{\mathrm{e}} \quad \triangleq \\ &\quad ([\![E_1]\!]^{\mathrm{e}} \implies [\![E_2]\!]^{\mathrm{e}}) \wedge ([\![\mathbf{not}(E_1)]\!]^{\mathrm{e}} \implies [\![E_3]\!]^{\mathrm{e}})\end{aligned}$$

match expressions are encoded similarly: for each branch, we ∀-quantify the matched constructor's arguments and relate them to the matched value by the constructor's relational abstraction, as a guard for the branch's body. Recall that, since we have already performed normalization and partial evaluation, the body of each if-then-else branch and **match** branch is **Prop**-type.

5.2 Decidability

Our encoding preserves the decidability logic of the type system's quantifier interaction trees. Specifically, an expression with a decidable quantifier interaction tree is guaranteed to be encoded into the well-known, decidable *Extended EPR* fragment of logic.

Definition 5.1 (Extended EPR). A first-order logic formula φ is in the *Extended EPR* fragment if its *sort graph* is acyclic. The sort graph's nodes are the sorts that appear in φ. A directed edge between D_1 and D_2 exists if, after φ is placed in negation-normal form, the term $\exists x : D_1.\ \varphi_1$ appears, such that φ_2 includes $\forall y : D_2.\ \varphi_3$. The edge also exists if a function symbol with type $D_1 \to D_2$ appears in φ.

Our encoding eliminates all function symbols, so we only need to consider the rule for sort edges that come from quantifier alternations. Intuitively, our encoding preserves the decidability argument of the type system because, in the expression that is being encoded, each **forall** and **exists** term is reflected in the quantifier alternation tree by a corresponding ∀ or ∃. Likewise, each

op term is reflected in the tree by a $\mathcal{F}$ term. The encoding turns the **op** into a $\forall$, which is how $\mathcal{F}$ is interpreted when considering the decidability of the tree.

Theorem 5.2 (Encoding is EPR). *If a module is decidable (Def. 3.4), then the encodings of its assertions are all in the Extended EPR fragment of logic.*

5.3 Agreement with Semantics

Our verification encoding is sound with respect to the partial-function semantics for the HOL interpretation of Λ_{EPR}, and it is also complete when higher-order primitive operators are not used. To demonstrate this, we first note that assignments of symbols to partial functions in HOL and assignments of symbols to relational abstractions of functions in FOL are isomorphic when symbols are not higher-order.

Definition 5.3 (First-order symbol assignment bijection). For a symbol with type $\mathbf{Data}(D_1) \to \cdots \to \mathbf{Data}(D_n) \to \mathbf{Data}(D)$, suppose a HOL model assigns it to a partial function f. Then we define the corresponding relationally-abstracted FOL model with the same domains for $D_1, \ldots, D_n, D$, in which the symbol is assigned to a relation R for which $R(V_1, \ldots, V_n, V)$ iff $f(V_1, \ldots, V_n) = V$. We can likewise go in the opposite direction.

Theorem 5.4. *If our primitive function operators are restricted to first-order, then, given a partial-function HOL model M, with corresponding FOL model M^F, and a module L, the module is valid in M iff all FOL-encoded assertions are valid in M^F.*

When we consider higher-order function symbols, we can still define a mapping of HOL models into FOL models (but not the other way around).

Definition 5.5 (Higher-order symbol assignment mapping). For a symbol with type $T_1 \to \cdots \to T_n \to \mathbf{Data}(D)$, suppose a HOL model assigns it to a partial function f. Then we define the corresponding relationally-abstracted FOL model with the same domains for each of the first-order types among $T_1, \ldots, T_n, \mathbf{Data}(D)$. We assign the symbol to a relation R which takes the same first-order arguments, but in place of the higher-order arguments, takes values from some placeholder sort. For arguments $V_1, \ldots, V_n$, if $f(V_1, \ldots, V_n) = V$, then we define R to hold for any choice of $R(V_1', \ldots, V_n', V)$, where the higher-order arguments have been replaced by arbitrary elements from the placeholder sort.

With this mapping, we can show that the existence of a counterexample to the HOL semantics implies the existence (via the mapping) of a counterexample to FOL.

Theorem 5.6 (Higher-order encoding soundness). *In the presence of higher-order primitive function operators, if a model M makes E false in the HOL semantics, then the corresponding FOL model M^F gives E false as well.*

6 Evaluation

In this section, we demonstrate the applicability of Λ_{EPR} to implementation and verification of programs in three domains: recursively defined operations on inductive datatypes, manipulation of heap-allocated linked lists, and distributed coordination. We have implemented a Rust verification tool, called Ravencheck, that uses Λ_{EPR} as its intermediary language. Ravencheck interacts with Rust source code using Rust's macro system and can be installed via Rust's package manager (cargo) without needing to patch the compiler.[3] Ravencheck takes programs written in a purely-functional subset of the Rust and annotated with *attributes* [The Rust Project Developers [n. d.]], and converts

[3]The tool is available on Rust's official package repository, crates.io [Lewchenko and Contributors 2026].

Table 1. Verification results for TIP benchmarks on inductively-defined natural numbers. *# Lemmas* is the number of lemma annotations that were verified to support the main property. *# Instantiations* is the number of values that we needed to explicitly instantiate for verification. *Verif. time* is the number of seconds that verification took. The operator == is a recursively defined relation, while = is the innate SMT identity relation.

	Property	# Lemmas	# Instantiations	Verif. time (s)
# 6	$n - (n + m) = Z$	0	0	0.07
# 7	$(n + m) - n = m$	0	0	0.06
# 8	$(k + m) - (k + n) = m - n$	7	1	0.99
# 9	$(a - b) - c = a - (b + c)$	4	2	3.71
# 10	$m - m = Z$	0	0	0.03
# 17	$(n \leq Z) = (n == Z)$	0	0	0.04
# 18	$i < S(i + m)$	0	0	0.07
# 21	$n \leq (n + m)$	0	0	0.04
# 22	$\mathtt{max}(\mathtt{max}(a, b), c) = \mathtt{max}(a, \mathtt{max}(b, c))$	0	0	10.13
# 23	$\mathtt{max}(a, b) = \mathtt{max}(b, a)$	0	0	0.08
# 24	$(\mathtt{max}(a, b) == a) = (b \leq a)$	0	0	0.15
# 25	$(\mathtt{max}(a, b) == b) = (a \leq b)$	0	0	0.15
# 31	$\mathtt{min}(a, \mathtt{min}(b, c)) = \mathtt{min}(\mathtt{min}(a, b), c)$	0	0	2.21
# 32	$\mathtt{min}(a, b) = \mathtt{min}(b, a)$	0	0	0.07
# 33	$(\mathtt{min}(a, b) == a) = (a \leq b)$	0	0	0.16
# 34	$(\mathtt{min}(a, b) == b) = (b \leq a)$	0	0	0.14
# 54	$(m + n) - n = m$	3	2	0.25
# 65	$i < S(m + i)$	2	0	0.39
# 69	$m \leq (n + m)$	2	0	0.12
# 70	$m \leq n \implies m \leq S(n)$	0	0	0.05
# 79	$(S(a) - b) - S(c) = (a - b) - c$	5	0	5.11

them internally to Λ_{EPR} for decidable verification. We use the CVC5 solver [Barbosa et al. 2022] for deciding EPR queries. All verification and performance experiments were run on a 2019 Dell XPS laptop with an Intel Core-i5 CPU at 1.60GHz and 8G RAM.

6.1 TIP Benchmarks

TIP [Claessen et al. 2015] is a collection of verification problems involving inductive datatypes and recursively defined operations upon them. These problems are natural to express in Λ_{EPR}, since they are purely functional, but verification does require developer effort. For example, verifying property #69 in Table 1 required us to identify and verify two lemmas: (1) the reflexivity property for the recursively-defined $\leq$ relation, and (2) successor monotonicity ($a \leq b \implies a \leq S(b)$).

Table 1 summarizes the results of applying Ravencheck to a subset of the TIP problems: the 21 problems in the IsaPlanner set that exclusively involve natural numbers. Note that, while the properties all fall within the decidable theory of linear arithmetic, the TIP challenge is to verify them for an inductive definition of Nat, with recursive definitions for its operations ($+, -, \leq, ==$), rather than using an axiomatized theory in SMT. Accordingly, we use induction on natural numbers (defined and axiomatized in Ravencheck) to verify the properties listed in Table 1. The details of our induction method are available in the appendix.

Of the 21 problems, 15 required no lemmas or instantiations: this means that verifying them required no effort beyond defining the operations and property in the Rust syntax of Λ_{EPR}. The remaining 6 properties required lemmas in order to verify, and 3 of those required explicit quantifier

instantiations. The TIP benchmark proofs were completed by one (new) student over three weeks. This time included learning how to use Ravencheck, which was new to them.

6.2 Heap Reachability

Reachability is key to formal specifications for algorithms that manipulate heap structures made out of linked pointers. Reachability is a complex judgment, since it defines the transitive closure of the *next* function that connects one pointer to another. Despite this, prior work [Itzhaky et al. 2014b, 2013] has shown that reachability assertions on heap-manipulating programs can be verified using conditions encoded into EPR, as long as the assertions are expressed in a restricted fragment of logic. This fragment abstracts the transitive-closure definition of reachability, replacing it with an axiomatized, uninterpreted relation. The step function itself is a `Pointer` → `Pointer` function, and so it also must be abstracted out of the encoded conditions. The resulting logic is domain-specific, and carries a domain-specific semantics that must be understood in order to effectively use it for verification. We found that this logic can be encoded as an instance of our general language Λ_{EPR}, and we have used this result to verify the `reverse` function case-study from Itzhaky et al. [2013].

It may be surprising that Λ_{EPR}, as a purely-functional language, is equipped to replicate this prior work. Both the problem and approach of Itzhaky et al. [2013] are distinctly *imperative*: their verified programs make efficient use of loops and state mutation, and their reachability logic is based on weakest-precondition rules. However, we have implemented and verified a purely-functional `reverse` program that matches the performance of the original loop-based program. We accomplished this by using two well-known compilation techniques in our compilation of Rust-embedded Λ_{EPR} programs: function inlining and tail-call optimization. We expressed our Λ_{EPR} version of the reachability logic using axioms on primitive heap-updating functions, which relate the functions' inputs and outputs.

Implementation. To illustrate, consider our internal Rust implementation of the `set_next` operation, which we assume as a trusted primitive. Here, `Ptr` is a struct that wraps a `usize` array index. We assume that any index given as a `Ptr` falls within the heap's static range and is not equal to the `NULL_PTR` constant. We use the enum `Opt` to represent a possibly-null pointer (see Fig. 6b for type definitions).

```
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} } }
```

This function is declared as uninterpreted to our Λ_{EPR} verifier, and so it only needs to be *externally* pure: the body can use arbitrary Rust code as long as there are no side-effects visible beyond it.

As an externally-pure function, `set_next` must accept the entire heap it manipulates as an input, and return an entire new heap as an output. That sounds expensive, but the `Heap` type here is actually just a unique reference (`Box`) to a heap-allocated array (again, these details are opaque to the Λ_{EPR} verifier). The body of `set_next` simply mutates one cell in that array—the exact same action that the unverified `idiomatic_reverse_loop` function (Fig. 6a) performs—and then passes the unique reference back as its output. External code can only observe the change by inspecting the output `Heap`, which is assumed semantically to be an entirely new value. The Rust borrow-checker keeps the reference unique: the original `Heap` value is considered destroyed after being passed into `set_next`, and inspecting or reusing it would generate a compiler error. When compiled, the `set_next` call is *inlined*: it is replaced by its body, without the overhead of a function-call, so that the line `let h2 = set_next(c, d, h);` in Fig. 6b becomes a simple conditional, with each branch equivalent to the corresponding line `h[c] = d;` in Fig. 6a.

The `idiomatic_reverse_loop` function in Fig. 6a is a faithful reconstruction of the imperative list reverse program verified in [Itzhaky et al. 2013]. Since our tool does not have the ability to

```rust
type Heap = Box<[usize; HEAP_MAX]>;

pub fn idiomatic_reverse_loop(
    mut c: usize, mut h: Heap
) -> (usize, Heap) {
    let mut d = NULL_PTR;
    while c != NULL_PTR {
        let c2 = h[c];
        h[c] = d;
        d = c;
        c = c2;
    }
    (d,h)
}
```

(a) Idiomatic reverse function using a while-loop and mutable state.

```rust
struct Ptr(usize);
type Opt = Option<Ptr>;
type State = (Opt, Opt, Heap);

fn init(c: Opt, h: Heap) -> State {
    (c, None, h)
}
fn cond((c, d, h): State) -> Opt { c }
fn step(
    c: Ptr, (_, d, h): State
) -> State {
    let c2 = get_next(c, &h);
    let h2 = set_next(c, d, h);
    let d2 = Some(c);
    (c2,d2,h2)
}
fn finish((_, d, h): State) -> (Opt, Heap) {
    (d,h)
}
```

(b) Verifiable reverse function components, using a purely-functional subset of Rust.

Fig. 6. Two definitions of a linked-list reversal function. Definition (b) consists of several components that are assembled by the templates given in Fig. 7.

reason about loops, we first define and verify a tail-recursive function, and then generate its tail-call optimization—a semantically-equivalent loop—for runtime use. We found that the Rust compiler rarely performs automatic tail-call optimization,[4] and so we have implemented this process within Ravencheck. The user defines the loop component functions in Fig. 6b (`init`, `cond`, `step`, and `finish`), and then Ravencheck automatically constructs Fig. 7a and Fig. 7b, using the first for verification and passing the second to Rust for compilation.

Verification. We axiomatize the heap domain using a relation $\texttt{reaches}(x, y, h)$, meaning that a path exists in $h : \texttt{Heap}$ of connected `next` pointers from $x : \texttt{Ptr}$ to $y : \texttt{Ptr}$. We write this as $x \rightarrow_h y$ for short. When the call $\texttt{set\_next}(x, \texttt{Some}(y), h_1)$ produces h_2, our axiom states that a path $a \rightarrow_{h_2} b$ exists iff one of the following conditions is met: (1) a path $a \rightarrow_{h_1} b$ exists that does not cross x, or (2) $a \rightarrow_{h_1} x$ and $y \rightarrow_{h_1} b$ exist. When $\texttt{set\_next}(x, \texttt{None}, h_1)$ produces h_2, our axiom states that $a \rightarrow_{h_2} b$ exists iff the first condition is met.

Our goal specification for a reversed list is that a path exists in the input list iff the flipped path exists in the output list. For input heap `h0`, input list head `c0`, output heap `hf`, and output list head `df`, we write this as:

```
forall(|x: Ptr, y: Ptr| { for_root(c0, h0, |root: Ptr| {
    (reach(df,y,hf) && reach(y,x,hf))
        == (reach(c0,x,h0) && reach(x,y,h0))
}) })
```

Here, `for_root` universally-quantifies a `root` pointer that is reached by `c0` and is terminal, so that the body is only required to hold when the input list is acyclic.

[4]Rust relies on LLVM for tail-call elimination. An explicit tail call elimination feature is currently under development.

```
fn compiled_reverse_rec(
    c: Opt, h: Heap
) -> (Opt, Heap) {
    let s0 = init(c, h);
    // Recursive stepper replaces loop
    let s = step_rec(s0);
    finish(s)
}
fn step_rec(s: State) -> State {
    match cond(s) {
        Some(c) => {
            let s2 = step(c, s);
            step_rec(s2)
        }
        None => s,
    }
}
```

(a) Tail-recursive function compiled from the defined components.

```
fn compiled_reverse_loop(
    c: Opt, h: Heap
) -> (Opt, Heap) {
    let mut s = init(c, h);
    // Loop replaces recursive stepper
    while let Some(c) = cond(s) {
        s = step(c, s);
    }
    finish(s)
}
```

(b) Loop-based function compiled from the defined components.

Fig. 7. Two semantically equivalent reverse functions, assembled from the components defined in Fig. 6b. The loop-based function (b) is the tail-call optimization of the recursion-based function (a).

We were able to verify `simple_reverse_rec`, a non-tail-recursive reverse function (definition available in our artifact), simply by checking this specification as an inductive annotation. To verify the `compiled_reverse_rec` function (and thus `compiled_reverse_loop`), we used an additional annotation on `rec_step`, akin to a loop invariant:

```
forall(|x: Ptr, y: Ptr| { for_root(cn, hn, |root: Ptr| {
    !reach(dn,root,hn) =>
    (reach(df,y,hf) && reach(y,x,hf))
        == ((reach(cn,y,hn) && reach(y,x,hn)) || (reach(dn,x,hn) && reach(x,y,hn))
            || (reach(cn,x,hn) && reach(dn,y,hn)))
}) })
```

Here, `cn` is the head of the input list fragment that remains, and `dn` is the head of the reversed list fragment that has been built so far. We require that the `dn` list does not reach the root of the `cn` list, and thus does not include any nodes from `cn`.

Our specification and additional annotation are essentially equivalent to the specification and inductive invariant verified for the `reverse` function in Itzhaky et al. [2013], though we make significantly weaker assumptions: (1) we do not assume that the entire heap is acyclic, and (2) we do not assume that the entire heap is contained within the list. Our `simple_reverse_rec` verification took 0.81*s*, and `compiled_reverse_rec` took 28.24*s*.

Performance. The goal of our tail-call-optimization compilation strategy was to verify a reverse function that matches the performance of `idiomatic_reverse_loop`. To evaluate this, we measured the execution time of `idiomatic_reverse_loop` and our three verified Ravencheck functions—the semantically equivalent pair `compiled_reverse_rec` and `compiled_reverse_loop` (Fig. 7), and the non-tail-recursive `simple_reverse_rec`—while varying the input list length. The results are plotted in Fig. 8. We found that `idiomatic_reverse_loop` and `compiled_reverse_loop` had nearly indistinguishable performance: they traded leads, remaining within 10 microseconds of each other on each list size. In contrast, `compiled_reverse_rec` function had slower performance,

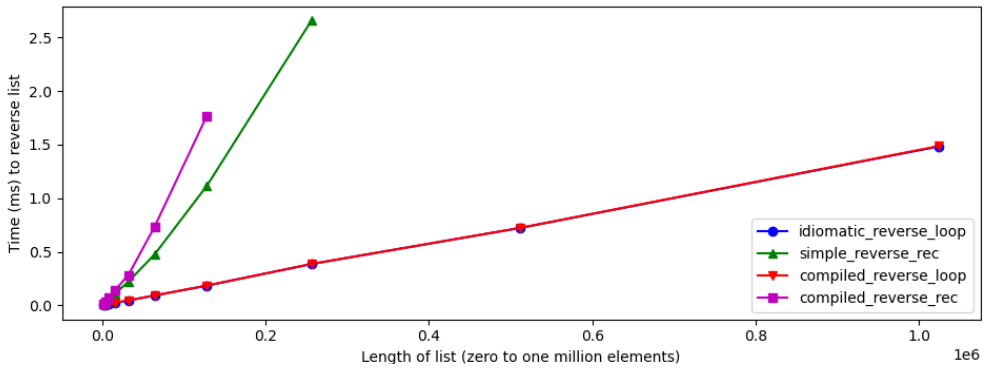

Fig. 8. Results of performance experiment for four different linked-list reverse functions. The two recursive functions, `compiled_reverse_rec` and `simple_reverse_rec`, encountered stack-overflows when attempting to process lists of size 256K and 512K, respectively. The two loop-based functions have nearly indistinguishable performance, varying by less than 10 microseconds for each list size.

and encountered a hard stack-overflow limit on the list of size $265K$. The `simple_reverse_rec` function actually outperformed `compiled_reverse_rec`, but was ultimately limited, compared to the loops, in a similar way.

These results demonstrate our ability to match the performance of idiomatic code when executing an Ravencheck-verified heap manipulation program. The performance difference between `compiled_reverse_loop` and `simple_reverse_rec` demonstrates a trade-off between runtime performance and verification effort: `simple_reverse_rec` required no additional annotations to verify, but it was less capable at runtime.

6.3 Distributed Coordination

To evaluate the scalability of verification with Λ_{EPR}, we applied it to Raft, a distributed coordination algorithm [Ongaro and Ousterhout 2014]. The Raft algorithm, which enables a distributed network of nodes to maintain a shared log of transactions while tolerating network failures, is a common non-trivial benchmark for verification frameworks. For example, the Rocq-based Verdi framework has been used to verify safety for an implementation of Raft—a manual effort involving a 50 000 line proof [Woos et al. 2016]. Using the Ivy framework, Taube et al. [2018] demonstrated that decidable SMT-based automation could significantly reduce effort, verifying a Raft implementation using just 300 lines of proof code. In both cases, *state machine safety* was verified[5]—the property that once a node has a committed entry in its log, then no node can ever have a different entry committed for the same index.

In our case study, we reduced the effort necessary to verify state machine safety by using two major differences in approach. First, we took advantage of Λ_{EPR}'s ability to share code between implementation and specification, as seen in the Fig. 9 code snippet: `committed`, an executable function, is used directly as the basis of the quantified safety specification `commit_safety`. In the Ivy implementation of Taube et al. [2018], an equivalent relation to `committed` is maintained as an uninterpreted ghost relation, which is kept up-to-date by bookkeeping code added to each message handler, and cannot be used at runtime. Because the `filter` function, used in the definition

[5]The Verdi project also verified linearizability for a client-request protocol atop Raft, but the authors noted that state machine safety accounted for the "vast majority" of the proof effort [Woos et al. 2016].

```
fn committed(t: Term, i: Index, s: &State) -> bool {
    is_quourum_of(
        filter(|v| up_to((t,v), i, &s.accepts), s.voters),
        s.voters)}
fn commit_safety() -> bool {
  forall(|s1: State, m: Msg, t: Term, i: Index| {
        let s2 = handle(m, s1.clone());
        committed(t, i, &s1) => prefix_match(i, &s1.log, &s2.log)}) }
```

Fig. 9. Safety specification for our Λ_{EPR} implementation of Raft consensus.

of `committed`, has voter-sets as both input and output, it must be relationally abstracted—thus `committed`, as defined in Fig. 9, could not be used within a quantified expression in Ivy.

Second, we implementated Raft consensus using the *Ferry* variant introduced by Lewchenko et al. [2025], which recasts Raft as an application of a causally-consistent replicated store. Using the *stable update precondition* proof theory of Lewchenko et al. [2025], we proved state machine safety for our implementation by defining network-stable preconditions for replicated updates, and then verifying precondition-non-interference and commutability conditions for each pair of updates. In this way, our verification conditions were focused directly on executable code: the update handlers that run on individual nodes at runtime. In contrast, the Verdi and Ivy projects both verified safety as a global invariant on a whole-network model. The correspondance of the model's transitions to the node-local executable code then also needed to be verified. It is important to note that Ferry, by relying on a causal-delivery network layer, has a larger trusted codebase than the implementations of the Verdi and Ivy projects, which assume no guarantees on message delivery order. In principle, a separately-verified causal-delivery implementation could be used, such as that of Redmond et al. [2023].

Our implementation consists of 261 lines of code, of which 110 are specification and verification artifacts, and verification took about 50 minutes. We performed this case study using our original, Haskell-based prototype of the Λ_{EPR} language: the code in Fig. 9 corresponds to equivalent Haskell-syntax functions in our implementation, available in our paper's artifact. To demonstrate the executability of our implementation, we compared its performance to `etcd` [etcd 2024]—a widely-used, nonverified Raft implementation—using a cluster 5 nodes running in a simulated network on a single laptop machine. At a request-rate of 10/ms, our Ferry implementation had 9.8/ms average throughput and 43.4 ms average latency. Under the same conditions, `etcd` demonstrated 9.9/ms average throughput and 32.5 ms average latency.

7 Related Work

Proof Assistants. Interactive theorem provers, such as Rocq, Lean, and Isabelle have been the primary choice for large-scale software verification efforts, such as CompCert [Leroy 2009], CakeML [Kumar et al. 2014], VST [Appel 2011], Iris [Jung et al. 2015], and Sel4 [Klein et al. 2010]. These tools assume a small trusted code base and enable *foundational* verification from first principles, generating machine-checkable proof objects as certificates. Proof theory is general-purpose and deduction in higher-order logic is supported. However, such generality comes at the expense of automation: verification is mostly driven by hand-written proofs and the proof effort often exceeds the development effort, sometimes by an order of magnitude. Unlike proof assistants, Ravencheck's proof theory is based on a restricted fragment of higher-order logic that is amenable to predictable automation. Ravencheck is therefore strictly less general than proof

assistants. Verification is driven by an SMT solver searching for a satisfiable model and no proof certificate is generated.

SMT-aided Verification. Language-integrated verification frameworks, such as Dafny [Leino 2010], F* [Swamy 2023], and Liquid Haskell [Vazou et al. 2014], allow programs to be annotated with specifications in the form of dependent types or assertions in a program logic, e.g, Separation Logic. Applying the type rules or proof rules generates verification conditions, which are discharged using an SMT solver. Liquid Haskell's dependent type system restricts type refinements to quantifier-free fragment of logic to ensure the decidability of type checking. Quantified assertions need to be expressed as function types for which proof objects need to be built explicitly. F*'s dependent type system imposes no such restrictions and therefore incurs undecidability. Successful type checking often requires programmer intervention in the form of `SMTPat` annotations on quantifiers or `assert` statements asserting helper lemmas. Similar observations apply to Dafny, which is a Hoare logic-based verifier for imperative programs. Ravencheck, on the other hand, is based on higher-order logic (HOL). In this sense, Ravencheck is similar to Isabelle/HOL except that proofs are always SMT-aided. Sledgehammer [Böhme and Nipkow 2010] is a tactic extension to Isabelle that lets programmers use automated theorem provers (including SMT solvers) to discharge proof obligations of the current goal. The tactic might fail, in which case manual proof is required. In contrast, Ravencheck guarantees decidability of verification while admitting a restricted form of quantification and higher-order functions.

Verification by Reduction to EPR. As described in Sec. 6.2, [Itzhaky et al. 2014a] uses EPR logic to reason about heap reachability properties. Catalyst [Kaki and Jagannathan 2014] introduces a specification language based on EPR to describe the morphisms on functional data structures. Padon et al. [2017] automatically verifies a model of Paxos consensus algorithm by EPR encoding.

Ivy [McMillan and Padon 2020], a widely-known verification framework based on Extended EPR, is primarily a modeling language for state transition systems with an integrated verification framework that generates verification conditions (VCs) in EPR. Our language and implementation improve upon Ivy in several ways. First, Ivy checks for sort-cycles using a whole-program analysis on VCs, while Ravencheck uses a compositional type system. Second, Ivy distinguishes syntactically between imperative *actions* that are relationally-abstracted and logical *functions* that are total, and places different restrictions on each. Third, Ivy supports extraction to executable C++ code, but the significant difference between Ivy and C++ makes the intended execution behavior unclear to non-experts. In Ravencheck, quantifier-free code is already valid Rust code that can be compiled directly by the Rust compiler, making its execution semantics straightforward to understand.

8 Conclusion

Decidable, SMT-based automation shows promise as a way to bring software verification into mainsteam development practice. In this paper, we use Λ_{EPR} to overcome an obstacle to this approach: due to the gap in expressivity between the Extended EPR fragment of logic and familiar programming languages, the software developer must straddle those two different domains when performing verification.

By employing automatic relational abstraction and, crucially, explaining the effects of this transformation as a semantics of partial functions, Λ_{EPR} enables developers to write their programs and verification artifacts together in a familiar, functional language. The type system of Λ_{EPR} assists developers in navigating the quantifier restrictions that are needed for decidability. Our Ravencheck tool concretely applies this novel approach to the Rust language, and demonstrates a key benefit: by lifting the domain of verification into the familiar language of Rust, it becomes straightforward to efficiently integrate verified code with existing Rust projects and libraries.

Data-Availability Statement

Our artifact for this paper [Lewchenko et al. 2026] includes the prototype implementation of Ravencheck and the Ravencheck code that constitutes our case studies.

References

Andrew W. Appel. 2011. Verified software toolchain. In *Proceedings of the 20th European Conference on Programming Languages and Systems: Part of the Joint European Conferences on Theory and Practice of Software* (Saarbrücken, Germany) *(ESOP'11/ETAPS'11)*. Springer-Verlag, Berlin, Heidelberg, 1–17.

O. Arieli, A. Avron, and A. Zamansky. 2011. Ideal Paraconsistent Logics. *Stud. Log.* 99, 1–3 (Oct. 2011), 31–60. doi:10.1007/s11225-011-9346-y

Haniel Barbosa, Clark W. Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. 2022. cvc5: A Versatile and Industrial-Strength SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 13243)*, Dana Fisman and Grigore Rosu (Eds.). Springer, 415–442. doi:10.1007/978-3-030-99524-9_24

Henry Blanchette, Niki Vazou, and Leonidas Lampropoulos. 2022. Liquid proof macros. In *Proceedings of the 15th ACM SIGPLAN International Haskell Symposium* (Ljubljana, Slovenia) *(Haskell 2022)*. Association for Computing Machinery, New York, NY, USA, 27–38. doi:10.1145/3546189.3549921

Sascha Böhme and Tobias Nipkow. 2010. Sledgehammer: judgement day. In *Proceedings of the 5th International Conference on Automated Reasoning* (Edinburgh, UK) *(IJCAR'10)*. Springer-Verlag, Berlin, Heidelberg, 107–121. doi:10.1007/978-3-642-14203-1_9

Ergon Börger, Erich Grädel, and Yuri Gurevich. 1996. *The Classical Decision Problem*. Springer-Verlag Telos.

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.

etcd. 2024. etcd: a distributed reliable key-value store. https://github.com/etcd-io/etcd. Accessed: 2024-10-15.

William M. Farmer. 2008. The seven virtues of simple type theory. *Journal of Applied Logic* 6, 3 (2008), 267–286. doi:10.1016/j.jal.2007.11.001

Paul Hudak. 1989. Conception, evolution, and application of functional programming languages. *ACM Comput. Surv.* 21, 3 (Sept. 1989), 359–411. doi:10.1145/72551.72554

Shachar Itzhaky, Anindya Banerjee, Neil Immerman, Ori Lahav, Aleksandar Nanevski, and Mooly Sagiv. 2014a. Modular reasoning about heap paths via effectively propositional formulas. *SIGPLAN Not.* 49, 1 (Jan. 2014), 385–396. doi:10.1145/2578855.2535854

Shachar Itzhaky, Anindya Banerjee, Neil Immerman, Ori Lahav, Aleksandar Nanevski, and Mooly Sagiv. 2014b. Modular reasoning about heap paths via effectively propositional formulas. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Diego, California, USA) *(POPL '14)*. Association for Computing Machinery, New York, NY, USA, 385–396. doi:10.1145/2535838.2535854

Shachar Itzhaky, Anindya Banerjee, Neil Immerman, Aleksandar Nanevski, and Mooly Sagiv. 2013. Effectively-Propositional Reasoning about Reachability in Linked Data Structures. In *Proceedings of the 25th International Conference on Computer Aided Verification - Volume 8044* (Saint Petersburg, Russia) *(CAV 2013)*. Springer-Verlag, Berlin, Heidelberg, 756–772.

Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. 2015. Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Mumbai, India) *(POPL '15)*. Association for Computing Machinery, New York, NY, USA, 637–650. doi:10.1145/2676726.2676980

Gowtham Kaki and Suresh Jagannathan. 2014. A relational framework for higher-order shape analysis. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming* (Gothenburg, Sweden) *(ICFP '14)*. Association for Computing Machinery, New York, NY, USA, 311–324. doi:10.1145/2628136.2628159

Gerwin Klein, June Andronick, Kevin Elphinstone, Gernot Heiser, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. 2010. seL4: formal verification of an operating-system kernel. *Commun. ACM* 53, 6 (June 2010), 107–115. doi:10.1145/1743546.1743574

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.

Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. 2014. CakeML: a verified implementation of ML. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Diego, California,

USA) *(POPL '14)*. Association for Computing Machinery, New York, NY, USA, 179–191. doi:10.1145/2535838.2535841

K. Rustan M. Leino. 2010. Dafny: an automatic program verifier for functional correctness *(LPAR'10)*. Springer-Verlag, Berlin, Heidelberg, 348–370.

Xavier Leroy. 2009. Formal verification of a realistic compiler. *Commun. ACM* 52, 7 (July 2009), 107–115. doi:10.1145/1538788.1538814

Paul Blain Levy. 1999. Call-by-Push-Value: A Subsuming Paradigm. In *Proceedings of the 4th International Conference on Typed Lambda Calculi and Applications (TLCA '99)*. Springer-Verlag, Berlin, Heidelberg, 228–242.

Nicholas V. Lewchenko and Contributors. 2026. Ravencheck. https://crates.io/crates/ravencheck.

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

Nicholas V. Lewchenko, Kunha Kim, Bor-Yuh Evan Chang, and Gowtham Kaki. 2026. *Artifact for Effectively Propositional Higher- Order Functional Programming*. doi:10.5281/zenodo.18719029

Kenneth L. McMillan and Oded Padon. 2020. Ivy: A Multi-modal Verification Tool for Distributed Algorithms. In *Computer Aided Verification*, Shuvendu K. Lahiri and Chao Wang (Eds.). Springer International Publishing, Cham, 190–202.

Diego Ongaro and John Ousterhout. 2014. In Search of an Understandable Consensus Algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference* (Philadelphia, PA) *(USENIX ATC'14)*. USENIX Association, USA, 305–320.

Oded Padon, Giuliano Losa, Mooly Sagiv, and Sharon Shoham. 2017. Paxos Made EPR: Decidable Reasoning About Distributed Protocols. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 108 (Oct. 2017), 31 pages. doi:10.1145/3140568

Benjamin C. Pierce. 2002. *Types and Programming Languages* (1st ed.). The MIT Press.

Graham Priest. 1979. The Logic of Paradox. *Journal of Philosophical Logic* 8, 1 (1979), 219–241. http://www.jstor.org/stable/30227165

Patrick Redmond, Gan Shen, Niki Vazou, and Lindsey Kuper. 2023. Verified Causal Broadcast with Liquid Haskell. In *Proceedings of the 34th Symposium on Implementation and Application of Functional Languages* (Copenhagen, Denmark) *(IFL '22)*. Association for Computing Machinery, New York, NY, USA, Article 6, 13 pages. doi:10.1145/3587216.3587222

Nikhil Swamy. 2023. Proof-oriented Programming in F*. https://www.fstar-lang.org/tutorial/book/index.html.

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

Marcelo Taube, Giuliano Losa, Kenneth L. McMillan, Oded Padon, Mooly Sagiv, Sharon Shoham, James R. Wilcox, and Doug Woos. 2018. Modularity for Decidability of Deductive Verification with Applications to Distributed Systems. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Philadelphia, PA, USA) *(PLDI 2018)*. Association for Computing Machinery, New York, NY, USA, 662–677. doi:10.1145/3192366.3192414

The Rust Project Developers. [n. d.]. Attributes. The Rust Reference. https://doc.rust-lang.org/reference/attributes.html Accessed on: 2025-10-09.

Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon Peyton-Jones. 2014. Refinement types for Haskell. *SIGPLAN Not.* 49, 9 (Aug. 2014), 269–282. doi:10.1145/2692915.2628161

Philip Wadler. 1992. The essence of functional programming. In *Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Albuquerque, New Mexico, USA) *(POPL '92)*. Association for Computing Machinery, New York, NY, USA, 1–14. doi:10.1145/143165.143169

Doug Woos, James R. Wilcox, Steve Anton, Zachary Tatlock, Michael D. Ernst, and Thomas Anderson. 2016. Planning for Change in a Formal Verification of the Raft Consensus Protocol. In *Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs* (St. Petersburg, FL, USA) *(CPP 2016)*. Association for Computing Machinery, New York, NY, USA, 154–165. doi:10.1145/2854065.2854081

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)*. 178–188. doi:10.34727/2023/isbn.978-3-85448-060-0_26

Received 2025-10-09; accepted 2026-02-17