Automated Verification of Fundamental Algebraic Laws

GEORGE ZAKHOUR, University of St. Gallen, Switzerland
PASCAL WEISENBURGER, University of St. Gallen, Switzerland
GUIDO SALVANESCHI, University of St. Gallen, Switzerland

Algebraic laws of *functions in mathematics* – such as commutativity, associativity, and idempotence – are often used as the basis to derive more sophisticated properties of complex mathematical structures and are heavily used in abstract computational thinking. Algebraic laws of *functions in coding*, however, are rarely considered. Yet, they are essential. For example, commutativity and associativity are crucial to ensure correctness of a variety of software systems in numerous domains, such as compiler optimization, big data processing, data flow processing, machine learning or distributed algorithms and data structures. Still, most programming languages lack built-in mechanisms to enforce and verify that operations adhere to such properties.

In this paper, we propose a verifier specialized on a set of fundamental algebraic laws that ensures that such laws hold in application code. The verifier can conjecture auxiliary properties and can reason about both equalities and inequalities of expressions, which is crucial to prove a given property when other competitors do not succeed. We implement these ideas in the Propel verifier. Our evaluation against five state-of-the-art verifiers on a total of 142 instances of algebraic properties shows that Propel is able to automatically deduce algebraic properties in different domains that rely on such properties for correctness, even in cases where competitors fail to verify the same properties or time out.

CCS Concepts: • **Theory of computation** → *Program verification*; *Type structures*; • **Computing methodologies** → *Theorem proving algorithms.*

Additional Key Words and Phrases: Algebraic Properties, Type Systems, Verification

ACM Reference Format:
George Zakhour, Pascal Weisenburger, and Guido Salvaneschi. 2024. Automated Verification of Fundamental Algebraic Laws. *Proc. ACM Program. Lang.* 8, PLDI, Article 178 (June 2024), 24 pages. https://doi.org/10.1145/3656408

1 INTRODUCTION

Fundamental algebraic laws such as commutativity, associativity, and idempotence are routinely used at all levels – from elementary school arithmetics to abstract algebra – to reason about the properties of functions *in mathematics*. Such prominence, however, is hardly reflected in functions *in coding*, where algebraic properties are almost never explicit. Yet, these basic algebraic laws are highly important in a number of computing domains. For example, in compilers, commutativity enables optimizations based on code reordering [25, 34, 39]. In big data and stream processing systems, it allows parallel execution on independent nodes [13]. In high-performance computing, commutativity of operations improves the efficiency of reductions [27]. In distributed systems, it ensures that changes can be applied in any order and replicas eventually converge to the same

Authors' addresses: George Zakhour, University of St. Gallen, St. Gallen, Switzerland, george.zakhour@unisg.ch; Pascal Weisenburger, University of St. Gallen, St. Gallen, Switzerland, pascal.weisenburger@unisg.ch; Guido Salvaneschi, University of St. Gallen, St. Gallen, Switzerland, guido.salvaneschi@unisg.ch.

ACM 2475-1421/2024/6-ART178
https://doi.org/10.1145/3656408

value [41]. In abstract models, it ensures that models can be merged [12, 35, 36]. In databases, it enables concurrency control [49].

Unfortunately, modern programming languages lack abstractions that allow developers to reason about and ensure common algebraic properties of operations. While these fundamental properties are constantly referred to in the mathematical domain, in software, this lack persists even though algebraic properties of functions are critical for various crucial correctness guarantees. Instead, developers are left with the responsibility of informally documenting functions with their supposed properties, placing the burden on developers to ensure that functions actually satisfy these properties. The absence of any form of enforcement for algebraic correctness can potentially result in erroneous program behavior and significantly degrade system performance or reliability. For example, the Message Passing Interface (MPI) function `MPI_OP_CREATE(function, commute, op)` creates an operation that can be passed to a reduce or a scan. The `commute` parameter is a boolean value that specifies whether the function commutes; further, the operation is always assumed to be associative [27]. Yet, this assumption is neither statically nor dynamically checked.

This paper addresses this gap by proposing a novel approach for the core of the automated Propel verifier [53], targeting a set of fundamental algebraic properties of operations and relations relevant to many application domains. For example, our approach enables the operation passed to `MPI_OP_CREATE` to be statically checked to be associative. Our key insight is Propel's approach to discover auxiliary properties and the way it can reason about both equalities and inequalities. These features enable the derivation of contradictions from both, giving Propel a crucial advantage over existing verifiers.

In the evaluation we consider five state-of-the-art competitors and compare their ability to verify algebraic properties in different domains, investigating a total of 142 properties. The results indicate that Propel can prove a large number of relevant instances of algebraic operations lawful, outperforming existing approaches that do not feature dedicated support for algebraic reasoning.

This paper makes the following contributions:

- We present a novel design of Propel, a domain-specific theorem prover specialized in algebraic properties of functions that can reason about both equalities and inequalities.
- We formalize Propel based on a core calculus. We prove the verifier sound, i.e., we show that derived algebraic properties for functions hold.
- We implement Propel in Scala for proving terms in the Propel calculus.
- We evaluate Propel, demonstrating that it outperforms competing systems when verifying algebraic properties in terms of the amount of properties proven in different domains.

The paper is organized as follows: Section 2 informally introduces our approach. Section 3 presents Propel's core calculus for proving algebraic properties. Section 4 evaluates our approach and the impact of specific features. Section 5 covers the related work. Section 6 concludes.

2 LAWFULNESS OF OPERATIONS

Different algebraic structures require the operations defined on their carrier set to obey different algebraic laws. For example, the abelian semigroup $(\mathbb{N}, +)$ requires that the axioms $a + b = b + a$ and $a + (b + c) = (a + b) + c$ hold, i.e., addition on natural numbers is commutative and associative. Yet, that these properties hold is often just assumed rather than checked by the programming language. We address the issue of checked algebraic properties by building on two ideas.

First, when declaring the type of a function, developers can specify the properties that should hold for the function, e.g., commutativity or associativity. An automated inductive prover will deduce whether the annotated properties hold and reject the program if they do not. The prover is focused on equational reasoning on common algebraic properties.

Second, the proven properties are included in a function's type. (1) This enables the verification of properties of function compositions. To prove properties of the composed function, the prover can make use of the properties of the individual functions that are being composed, as the properties are available through the types of these functions. (2) Functions that require certain properties for one of their arguments, can state the properties in the argument's type, in which case only functions for which the properties were proven can be passed as argument. This enables composition of higher-order functions, for which algebraic properties are required.

2.1 The Propel Language by Example

Example add_0. We demonstrate our approach to verifying algebraic properties on the addition of natural numbers as Peano numbers $\mathbb{N}$ with constructors Z and S, implemented by $\texttt{add}_0$ in Listing 1.

Example add_1. In Propel, functions are annotated with properties which, akin to their domain and codomain, are essential parts of their type. We state that $\texttt{add}_1$ is both commutative and associative.

We also offer a Scala embedding using types of the form `P :=(A, A) =>: B` to express algebraic properties of binary functions. `P` denotes the function properties and `A` and `B` the types of the arguments and the result. The introduction form for Propel functions is `prop[FunctionType]` (or `prop.rec` for recursive functions). The following code snippet shows the $\texttt{add}_1$ function implemented in our Scala DSL, asserting that addition is commutative and associative:

```
enum ℕ:
  case Z; case S(pred: ℕ)

def add1(x: ℕ, y: ℕ) = prop.rec[(Comm & Assoc) := (ℕ, ℕ) =>: ℕ]: add1 =>
  case (Z, y) => y
  case (S(x), y) => S(add1(x, y))
```

We avoid the embedding's syntactic noise in Listing 1 using a more abstract notation.

Propel will create a property derivation tree as part of type-checking that constitutes the property's proof. As the function is recursive, Propel employs induction. The following tree, which is a subtree of the full typing derivation, shows that associativity is proven trivially by induction on x:

$$\dfrac{\dfrac{}{\Gamma, x = Z \vDash \texttt{add}_1\ y\ z = \texttt{add}_1\ y\ z}\,(\textsc{Refl})}{\Gamma, x = Z \vDash \texttt{add}_1\ Z\ (\texttt{add}_1\ y\ z) = \texttt{add}_1\ (\texttt{add}_1\ Z\ y)\ z}\,(\textsc{Def}) \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma, w : \mathbb{N}, x = S\ w, \texttt{add}_1\ w\ (\texttt{add}_1\ y\ z) = \texttt{add}_1\ (\texttt{add}_1\ w\ y)\ z \vDash S\ (\texttt{add}_1\ w\ (\texttt{add}_1\ y\ z)) = S\ (\texttt{add}_1\ w\ (\texttt{add}_1\ y\ z))}\,(\textsc{Refl})}{\Gamma, w : \mathbb{N}, x = S\ w, \texttt{add}_1\ w\ (\texttt{add}_1\ y\ z) = \texttt{add}_1\ (\texttt{add}_1\ w\ y)\ z \vDash S\ (\texttt{add}_1\ w\ (\texttt{add}_1\ y\ z)) = S\ (\texttt{add}_1\ (\texttt{add}_1\ w\ y)\ z)}\,(\textsc{Hyp})}{\Gamma, w : \mathbb{N}, x = S\ w, \texttt{add}_1\ w\ (\texttt{add}_1\ y\ z) = \texttt{add}_1\ (\texttt{add}_1\ w\ y)\ z \vDash S\ (\texttt{add}_1\ w\ (\texttt{add}_1\ y\ z)) = \texttt{add}_1\ (S\ (\texttt{add}_1\ w\ y))\ z}\,(\textsc{Def})}{\Gamma, w : \mathbb{N}, x = S\ w, \texttt{add}_1\ w\ (\texttt{add}_1\ y\ z) = \texttt{add}_1\ (\texttt{add}_1\ w\ y)\ z \vDash \texttt{add}_1\ (S\ w)\ (\texttt{add}_1\ y\ z) = \texttt{add}_1\ (\texttt{add}_1\ (S\ w)\ y)\ z}\,(2\times\textsc{Def})$$

$$\dfrac{\cdot \vDash \underbrace{\forall x, y, z : \mathbb{N}}_{=\Gamma}.\ \texttt{add}_1\ x\ (\texttt{add}_1\ y\ z) = \texttt{add}_1\ (\texttt{add}_1\ x\ y)\ z}{}\,(\textsc{Ind})$$

The tree is a complete but slightly simplified version of the one Propel follows, employing induction (Ind). Its left subtree is the base case which applies the function definition (Def) and concludes by reflexivity (Refl). Its right subtree is the inductive case which applies (Def) three times, rewrites using the inductive hypothesis (Hyp), and concludes by (Refl).

Example add_2. However, a slight variation of the program, shown in $\texttt{add}_2$, hinders the simple proof of associativity. As the arguments to the recursive call are swapped, no induction hypothesis for $\texttt{add}_2$ x y can be immediately applied for the recursive call $\texttt{add}_2$ y x. Had Propel deduced that $\texttt{add}_2$ is commutative then it could rewrite $\texttt{add}_2$ y x to $\texttt{add}_2$ x y and apply the induction hypothesis to prove associativity. Generally, the proofs for some properties rely on other properties. Section 2.3 elaborates on the algebraic properties our prover supports and the order in which we prove them.

Example add_3. Function $\texttt{add}_3$ adopts yet another formulation for addition. $\texttt{add}_3$'s associativity is challenging to prove automatically for many state-of-the-art verifiers (as shown in Section 4).

Listing 1. Addition with Peano Numbers, four versions.

```
let rec add₀
  = λ x y. cases x of
      Z   ⇒  y
      S x ⇒  S (add₀ x y)

let rec add₁: ℕ —comm,assoc→ ℕ → ℕ
  = λ^comm, assoc x: ℕ y: ℕ. cases x of
      Z   ⇒  y
      S x ⇒  S (add₁ x y)

let rec add₂: ℕ —comm,assoc→ ℕ → ℕ
  = λ^comm, assoc x: ℕ y: ℕ. cases x of
      Z   ⇒  y
      S x ⇒  S (add₂ y x)

let rec add₃: ℕ —comm,assoc→ ℕ → ℕ
  = λ^comm, assoc x: ℕ y: ℕ. cases (x, y) of
      (Z, y)      ⇒  y
      (x, Z)      ⇒  x
      (S x, S y)  ⇒  S (S (add₃ x y))
```

Proving associativity for add_3 is more involved because it requires knowing how the data constructor S can be moved outwards from the function's arguments, i.e., add_3 (S x) y = S (add_3 x y) and add_3 x (S y) = S (add_3 x y). To prove algebraic properties, we found it useful to enable the prover to explore how algebraic data type constructors of the function's arguments and its result relate. Section 2.4 discusses our approach to discover such auxiliary properties.

Type-checking add_3 and checking its commutativity property, on the other hand, is straightforward. Propel produces the following tree, which combines the typing and the property derivation:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{\underbrace{\text{add}_3 : \mathbb{N} \xrightarrow{\text{comm}} \mathbb{N} \to \mathbb{N},\ x : \mathbb{N}, y : \mathbb{N}}_{=\Gamma_0} \vdash \text{cases } (x,y) \text{ of } \ldots : \mathbb{N}}\ (T) \qquad \dfrac{\dfrac{\dfrac{\dfrac{\Gamma_1 \vdash \text{add}_3 : \mathbb{N} \xrightarrow{\text{comm}} \mathbb{N} \to \mathbb{N}}{\Gamma_1 \triangleright \Gamma_1, \text{add}_3\ a\ b = \text{add}_3\ b\ a}\ (P4) \quad \dfrac{x = S\ a \in \Gamma_1}{a \le x}\ (P5) \quad \dfrac{y = S\ b \in \Gamma_1}{b \le y}\ (P6)}{\Gamma_1 \Vdash S(S(\text{add}_3\ a\ b)) = S(S(\text{add}_3\ b\ a))}\ (P3)}{\underbrace{\Gamma_0, a : \mathbb{N}, b : \mathbb{N}, x = S\ a, y = S\ b}_{=\Gamma_1} \Vdash \text{add}_3\ (S\ a)\ (S\ b) = \text{add}_3\ (S\ b)\ (S\ a)}\ (P2)}{\vdots\ (P1)\ \Gamma_0 \Vdash \text{add}_3\ x\ y = \text{add}_3\ y\ x}}{\cdot \vdash \text{add}_3 : \mathbb{N} \xrightarrow{\text{comm}} \mathbb{N} \to \mathbb{N}}$$

The subtree (T) represents the standard typing rules. Yet, to complete the type checking, we must also prove commutativity. (P1) represents this proof using the $\Vdash$ relation. The first step is induction on the pair (x, y) as dictated by the case expression. We omit the base cases where either x or y are Z since they are trivial. (P2) is the proof of the inductive case, which reduces both sides of the equality (P3). Then, we deduce (represented by the $\triangleright$ relation) the commutativity property from the proof context (P4). The deduction is valid because the function being applied is assumed to have this property (inductive hypothesis) and its arguments are structurally smaller (P5) and (P6).

The derivation tree is based on the Propel core calculus, which is based on the simply-typed lambda calculus extended with (1) property annotations on lambdas and function types, (2) algebraic data types and (3) pattern matching on instances of algebraic data types. The Peano numbers are represented by the algebraic data type $\mathbb{N} := Z + S\ \mathbb{N}$, and the booleans by $\mathbf{2} := \top + \bot$. Section 2.6 provides insights into how the key components of the prover work algorithmically. The core language is presented in Section 3.

Example of function composition. Propel can prove properties of the composition of functions based on the properties of the individual functions. The following code defines a higher-order zipWith function that takes a commutative and associative function over some type T and returns a commutative and associative function over lists of T by applying the function pair-wise to the lists:

```
let rec zipWith: (T —comm,assoc→ T → T) → (List[T] —comm,assoc→ List[T] → List[T])
  = λf: T —comm,assoc→ T → T. λ^comm,assoc xs: List[T], ys: List[T]. cases (xs, ys) of
      (Nil, _) | (_, Nil) ⇒ Nil
      (Cons x xs, Cons y ys)  ⇒ Cons (f x y) (zipWith xs ys)
```

The algebraic properties of the argument function are required to prove the algebraic properties of the returned function. Moreover, zipWith can only be called with a function that the prover has verified to be commutative and associative.

In general, including algebraic properties in a function's type enables parametrization, i.e., a function does not need to refer to another function that has some algebraic properties directly but can instead use a function parameter that includes the properties in its type (e.g., the f parameter of zipWith). Propel can prove the function correct because it manipulates function calls by using the properties attached to the function (i.e., the prover does not unfold the definitions).

2.2 The Algebraic Properties in Propel

The algebraic properties supported by Propel fall into four categories:

- A predefined set of fundamental algebraic properties – as known from mathematics – that are both of interest in many domains and valuable for proving further properties – which are examined for every function (Section 2.3).
- A predefined set of auxiliary algebraic properties that are essential for proving the aforementioned properties – namely distributivity and unfolding of data constructors – which are examined before the fundamental algebraic properties (Section 2.4).
- Custom properties asserted on functions by the developer to be proven by Propel (Section 2.5).
- A set of equalities and inequalities of terms that are discovered while attempting a proof, which are not exposed to the developer but essentially represent the proof state and serve as rewrite rules (Section 2.6).

2.3 Propel's Fundamental Algebraic Properties

Propel focuses on a set of fundamental algebraic properties of functions important for a variety of algebraic structures, such as (abelian) (semi)groups, rings or (semi)lattices. Propel attempts to prove the algebraic properties in the following list for every function and include the properties in the function's type in case a proof can be found:

(Commutativity)	$f(x, y) = f(y, x)$	(Reflexivity)	$xRx = \top$
(Selection)	$f(x, y) = x \vee f(x, y) = y$	(Irreflexivity)	$xRx = \bot$
(Idempotence)	$f(x, x) = x$	(Antisymmetry)	$x \neq y \rightarrow xRy = \top \rightarrow yRx = \bot$
(Injectivity)	$f(x) = f(y) \rightarrow x = y$	(Symmetry)	$xRy = \top \rightarrow yRx = \top$
(Associativity)	$f(x, f(y, z)) = f(f(x, y), z)$	(Connected)	$x \neq y \rightarrow (xRy = \top \vee yRx = \top)$
		(Transitivity)	$xRy = \top \rightarrow yRz = \top \rightarrow xRz = \top$

Operations are of type $\mathsf{T} \rightarrow \mathsf{T} \rightarrow \mathsf{T}$ and relations are of type $\mathsf{T} \rightarrow \mathsf{T} \rightarrow 2$ (i.e., relations are as functions returning a boolean with data constructors $\top$ and $\bot$).

When a function is annotated with these properties, both sides of all (in)equalities in each property's definition must have the same type. The order in which these properties are checked is as presented above, column-wise. We fixed the order for performance, proving "simpler" properties first, and more "complex" ones (i.e., involving more quantified variables) later, which we found to work well in practice in our experiments. The only dependencies between properties we encountered in the experiments are: associativity may need commutativity and transitivity may need symmetry.

Tracking relational properties is important in order to reason about conditional statements. For example, the following definition of the maximum function can only be proven commutative if the less-than relation (le) is known to be antisymmetric and connected:

```
let rec max: ℕ → ℕ —comm,assoc,idem,sel→ ℕ → ℕ = λ^comm, assoc, idem, sel x: ℕ y: ℕ.
  if le x y then y else x
```

Similarly, to prove associativity it must be known that le is transitive.

2.4 Propel's Auxiliary Properties

Before attempting a proof of any algebraic property in Section 2.3, Propel attempts to prove certain auxiliary ones which are instrumental to proving the properties annotated in the type by developers. To this end, the prover conjectures an additional set of properties and iteratively tries to prove them until success or no progress is possible, i.e., the prover exhausted its search space (to mitigate diverging proof search, Propel restricts the space as described in Section 2.6, Algorithm 2). The proven conjectures can be used in the subsequent proofs. We describe our approach to construct the two kinds of conjectures the prover explores.

Distributivity. To capture the distributivity relations across different functions, we collect all functions g used in the body of a function f, which are closed over the same type. We then conjecture that one function distributes over the other, i.e., g x (f y z) = f (g x y) (g x z) and f x (g y z) = g (f x y) (f x z). Distributivity conjectures are used in the same way we use the conjectures of unfolding data constructors, which we describe next.

Unfolding data constructors. To discover more auxiliary properties of a function, the prover evaluates it with every argument of an algebraic data type unfolded at most once and generalizes the reduced expression. For example, Peano numbers are unfolded into x, Z, and S x for fresh x. Thus, passing Z and (S x) to add_1 (Listing 1) reduces to add_1 Z (S x) = S x. The left-hand side (LHS) of the equation represents the function applied to the arguments $f\ a_0 \ldots a_n$ and the right-hand side (RHS) is the maximally reduced function application. We employ the following two generalizations.

First, when the RHS aligns with one of the arguments on the LHS, we replace both with a variable. Hence, we create the generalized conjecture add_1 Z x = x.

Second, we examine all free variables in the LHS (of the from $f\ a_0 \ldots a_n$), and the RHS. We verify that (i) all free variables exclusively appear within a single argument a_i of f and (ii) a_i has the same type as the result of f. If (i) and (ii) hold, we generate (a) a LHS where all arguments except a_i are generalized to variables and (b) a RHS where an arbitrary subexpression e_j matching the result type of f is replaced by the the LHS with a_i replaced by e_j (i.e., $f\ a_0 \ldots e_j \ldots a_n$). Hence, we create the generalized conjecture add_1 y (S x) = S (add_1 y x). Following this scheme, we further conjecture add_1 x Z = x, which is essential in the commutativity proof of add_1.

Proof example. As described before, Propel generates a type checking and property checking derivation tree. The following tree demonstrates the check for add_1's commutativity using the aforementioned auxiliary property add_1 x Z = x:

$$\dfrac{\dfrac{\text{(T)}\ \dfrac{\vdots}{\underbrace{\mathsf{add}_1 : \mathbb{N} \xrightarrow{\mathsf{comm}} \mathbb{N} \to \mathbb{N},\ \mathsf{x} : \mathbb{N}, \mathsf{y} : \mathbb{N}}_{=\Gamma_0} \vdash \mathsf{add}_1\ \mathsf{x}\ \mathsf{y} : \mathbb{N}}}{\text{(L)}\ \dfrac{\text{(LP)}\ \dfrac{\vdots}{\underbrace{\Gamma_0 \Vdash \forall \mathsf{w} : \mathbb{N}.\ \mathsf{add}_1\ \mathsf{w}\ \mathsf{Z} = \mathsf{w}}_{=\text{aux}}} \qquad \text{(P1)}\ \dfrac{\text{(P2)}\ \dfrac{\text{(P3)}\ \dfrac{\text{aux} \in \Gamma_1 \qquad \Gamma_1 \vdash \text{aux}[\mathsf{w} \to \mathsf{y}]}{\Gamma_1, \mathsf{x} = \mathsf{Z} \Vdash \mathsf{y} = \mathsf{add}_1\ \mathsf{y}\ \mathsf{Z}}}{\Gamma_1, \mathsf{x} = \mathsf{Z} \Vdash \mathsf{add}_1\ \mathsf{Z}\ \mathsf{y} = \mathsf{add}_1\ \mathsf{y}\ \mathsf{Z}} \quad \cdots}{\underbrace{\Gamma_0, \text{aux}}_{=\Gamma_1} \Vdash \mathsf{add}_1\ \mathsf{x}\ \mathsf{y} = \mathsf{add}_1\ \mathsf{y}\ \mathsf{x}}}{\Gamma_0 \Vdash \mathsf{add}_1\ \mathsf{x}\ \mathsf{y} = \mathsf{add}_1\ \mathsf{y}\ \mathsf{x}}}{\cdot \vdash \mathsf{add}_1 : \mathbb{N} \xrightarrow{\mathsf{comm}} \mathbb{N} \to \mathbb{N}}$$

The (T) subtree is the usual typing tree, left out for brevity. The (L) subtree proves the commutativity property of add_1 as its type is annotated as such. The (LP) subtree states the conjecture derived earlier and proves it by induction on w. The (P1) subtree adds the conjecture to its proof context and proceeds to prove commutativity by induction. The inductive case is trivial and resembles that of add_3. The base case on the other hand is proven by first evaluating the right-hand side of the equality (P2) and observing that the stated equality is a specialization of the auxiliary property aux proven earlier with w substituted with y (P3). It is essential to observe that the equality stated in (P2) or (P3) cannot be proven by induction as that requires y to be quantified over.

2.5 Developer-Defined Properties

Propel allows developers to assert properties for functions as universally quantified equalities $\forall x_0 : T_0, \ldots, x_n : T_n.\ e = e'$, where e and e' are arbitrary Propel expressions whose equality is to be proven. For example, asserting the left and right identity laws for add_1 (Listing 1) is as follows:

property for add_1: **forall** x: ℕ. add_1 Z x = x
property for add_1: **forall** x: ℕ. add_1 x Z = x

Developer-defined properties are proven after auxiliary and fundamental algebraic properties (Sections 2.3 and 2.4). The proof strategy is the same for all kinds of properties (Section 2.6).

2.6 Verification Strategy

Our prover follows three steps. First, it conjectures potentially useful auxiliary properties (Section 2.4) and attempts to prove them. Second, it tries to prove all common algebraic properties, starting with the simpler ones (Section 2.3). The proven properties become part of the checked function's type. When they are not a subset of the ones annotated by the developer the prover rejects the program. Third, it checks the remaining properties the function is annotated with (Section 2.5). The prover rejects the program if it fails to find a proof for these properties.

The verification process examines each conjecture and property by checking whether the given equalities of the form $e_0 = e_1$ hold. To achieve this, we perform case analysis on the branches a function can take. When pattern matching yields multiple cases, each case creates a branch in the property derivation tree. Thus, Propel ensures that the property holds for every branch. In every branch, we verify that e_0 equals e_1 by applying rewrite rules to both sides. These rules consist of:

- The evaluation rules of the operational semantics extended to open terms excluding the evaluation of recursive calls,
- The algebraic laws of the supported algebraic properties (e.g., commutativity for a function f enables rewriting $f\ x\ y$ to $f\ y\ x$ for arbitrary terms x and y) and
- Rewriting a term x to a term y if the prover can establish that x and y are equal. Equality of terms can be discovered by case analysis or algebraic laws.

Algorithm 1 specifies the recursive syntax-directed rewrite rules for expressions (variables on Line 2, functions on Line 3, and data constructor applications on Line 4), applies β-reduction on Line 7, branches on the unique constructors on Line 10, and collects the equalities $\mathcal{E}^+$ and inequalities $\mathcal{E}^-$ on Lines 7, 12 and 13 that hold in every branch. $\mathcal{E}^+$ is extended for the cases that matched a pattern and $\mathcal{E}^-$ for the cases that did not. Thus, the result of the algorithm is a set containing for every branch a triple of a (partially) evaluated term and the (in)equality sets.

Algorithm 1 Term Evaluation

1: $[\![\mathsf{e}]\!] =$ **match** e
2: $\quad \mathsf{x} \Rightarrow \{(\mathsf{x}, \varnothing, \varnothing)\}$
3: $\quad \lambda \mathsf{x}{:}\mathsf{T}.\mathsf{e}_1 \Rightarrow \{(\lambda \mathsf{x}{:}\mathsf{T}.t, \mathcal{E}^+, \mathcal{E}^-) \mid (t, \mathcal{E}^+, \mathcal{E}^-) \in [\![\mathsf{e}_1]\!]\}$
4: $\quad \mathsf{K}\ \mathsf{e}_1 \cdots \mathsf{e}_n \Rightarrow \{(\mathsf{K}\ t_1 \cdots t_n, \bigcup_i \mathcal{E}_i^+, \bigcup_i \mathcal{E}_i^-) \mid (t_i, \mathcal{E}_i^+, \mathcal{E}_i^-) \in [\![\mathsf{e}_i]\!]\}$
5: $\quad \mathsf{e}_1\ \mathsf{e}_2 \Rightarrow$
6: $\quad\quad \{$ **match** t_1
7: $\quad\quad\quad \lambda \mathsf{x}{:}\mathsf{T}.t_1 \Rightarrow (t_1[\mathsf{x} \mapsto t_2],\ \{\mathsf{x} = t_2\} \cup \mathcal{E}_1^+ \cup \mathcal{E}_2^+,\ \mathcal{E}_1^- \cup \mathcal{E}_2^-)$
8: $\quad\quad\quad$ **otherwise** $\Rightarrow (t_1\ t_2, \mathcal{E}_1^+ \cup \mathcal{E}_2^+, \mathcal{E}_1^- \cup \mathcal{E}_2^-)$
9: $\quad\quad \mid (t_1, \mathcal{E}_1^+, \mathcal{E}_1^-) \in [\![\mathsf{e}_1]\!],\ (t_2, \mathcal{E}_2^+, \mathcal{E}_2^-) \in [\![\mathsf{e}_2]\!]\}$
10: $\quad \mathsf{e}_0\ \mathsf{case}\ \{\mathsf{K}_1 \Rightarrow \mathsf{e}_1 \cdots \mathsf{K}_n \Rightarrow \mathsf{e}_n\} \Rightarrow$
11: $\quad\quad \bigcup_i \{(t_i\ \mathsf{x}_1 \cdots \mathsf{x}_m,$
12: $\quad\quad\quad \{t_0 = \mathsf{K}_i\ \mathsf{x}_1 \cdots \mathsf{x}_m\} \cup \mathcal{E}_0^+ \cup \mathcal{E}_i^+,$
13: $\quad\quad\quad \bigcup_{j \neq i} \{t_0 \neq \mathsf{K}_j\ \mathsf{x}_{j,1} \cdots \mathsf{x}_{j,m} \mid \mathrm{arity}(\mathsf{K}_j) = m\} \cup \mathcal{E}_0^- \cup \mathcal{E}_i^-)$
14: $\quad\quad\quad \mid (t_0, \mathcal{E}_0^+, \mathcal{E}_0^-) \in [\![\mathsf{e}_0]\!],\ (t_i, \mathcal{E}_i^+, \mathcal{E}_i^-) \in [\![\mathsf{e}_i]\!],\ \mathrm{arity}(\mathsf{K}_i) = m\}$

$\mathcal{E}^+$ and $\mathcal{E}^-$ model Propel's approach to reason about equality of terms. Although the language does not provide a dedicated construct for equality checks, there are two ways to establish the equality of two expressions. First,

for pattern matching, unification binds variables to expressions, treating both variables and bound expressions as equal. Second, certain relational properties – reflexivity, irreflexivity, antisymmetry, and connectedness – determine equality or inequality. For instance, if a relation R is reflexive and $xRy = \bot$ is known in a specific branch, we can deduce that $x \neq y$ within the same branch. Hence, reflexive, irreflexive, antisymmetric and connected relations are key to lift the relation between two expressions into meta-level equality and inequality.

By taking into account the algebraic properties of relations, Propel uncovers additional information, i.e., whether expressions can be proven equal or unequal. A distinctive feature of Propel is that it explicitly tracks inequalities of expressions – not only equalities as competing approaches.

Algorithm 2 shows how Propel checks whether two expressions e_0 and e_1 are equal. On Lines 8 and 9 we call Algorithm 1 to reduce e_0 and e_1 to the set of terms t_i and t_j to which the different branches evaluate with the (in)equalities ($\mathcal{E}_i^+$, $\mathcal{E}_j^+$, $\mathcal{E}_i^-$, $\mathcal{E}_j^-$) that hold for the respective cases. We then follow these steps. (1) Rewrite the equations induced by algebraic laws in all (in)equalities sets $\mathcal{E}^+$ and $\mathcal{E}^-$ and terms t (RewriteAlgebraicLaws, Line 10). For example, knowing a function f is commutative enables rewriting $f\,x\,y$ to $f\,y\,x$. (2) Rewrite the equations of the equality set $\mathcal{E}^+$ (RewriteEqualities, Line 11). (3) Restrict the extent to which rewrites can expand trees in terms of node count, removing trees whose node count n grew by at least $8 + 1.1 \cdot n$ (Line 12). (4) Filter out all branches that have contradicting (in)equalities sets (Line 15). For example, contradictions within $\mathcal{E}^+$ (such as $x = \mathsf{Z}$ and $x = \mathsf{S}\ \mathsf{Z}$), contradictions between $\mathcal{E}^+$ and $\mathcal{E}^-$ (such as $x = \mathsf{Z}$ and $x \neq \mathsf{Z}$) or ill-founded equalities (such as $x = \mathsf{S}\ x$). And, (5) restrict the search to the top 256 terms (Line 16). We repeat these steps until reaching a fixed point, i.e., all equalities are applied, and no further (in)equalities are discovered. All numerical constants were chosen empirically as they perform well.

Algorithm 2 Equality Checking

```
1:  procedure Check(e1 = e2)
2:      T ← {(e1 = e2, ∅, ∅)}
3:      repeat
4:          T' ← T
5:          T ← ∅
6:          for all (t0 = t1, E+, E−) ∈ T' do
7:              T ← T ∪ {(ti = tj, Ei+ ∪ Ej+, Ei− ∪ Ej−) |
8:                       (ti, Ei+, Ei−) ∈ [[t0]],      ▷ Algorithm 1
9:                       (tj, Ej+, Ej−) ∈ [[t1]]}      ▷ Algorithm 1
10:             T ← { RewriteAlgebraicLaws(T) | T ∈ T }
11:             T ← { RewriteEqualities(T) | T ∈ T }
12:             T ← { (t0 = t1, E+, E−) | (t0 = t1, E+, E−) ∈ T
13:                                     ∧ |t0| < 8 + 1.1 · |e0|
14:                                     ∧ |t1| < 8 + 1.1 · |e1| }
15:             T ← { T | T ∈ T ∧ Consistent(T) }
16:             F_l ← { (t0 = t1, E+, E−) | (t0 = t1, E+, E−) ∈ T
17:                                     ∧ (|t0|) + (|t1|) < l }
18:                                                   ▷ Algorithm 3
19:             T ← F_l with l such that |F_l| < 256
20:     until T ≠ T'
21:     return if there exists a t, E+ and E− such that
22:             (t = t, E+, E−) ∈ T
```

Inequality tracking. Tracking inequalities often helps in finding a proof faster, as we show in Section 4. To illustrate, consider an argmax function that given two pairs returns the second component of the pair with the greater first component:

let rec argmax = λ^{comm} $(\mathsf{t}_1, \mathsf{v}_1)$ $(\mathsf{t}_2, \mathsf{v}_2)$.
  if le t_1 t_2 **then if** eq t_1 t_2 **then** tieBreaker v_1 v_2 **else** v_1 **else** v_2

In case of equality, a commutative and selective tie-breaking function picks the second component. Consider the commutativity proof for the specific case when le t_1 $\mathsf{t}_2 = \top$ and le t_2 $\mathsf{t}_1 = \bot$. Employing inequalities, as Propel does, requires fewer steps (*depth*) in the proof. By the antisymmetry of le and after pushing the second equality to the proof context, Propel can immediately deduce $\mathsf{t}_1 \neq \mathsf{t}_2$. In that case, the goal **if** eq t_1 t_2 **then** tieBreaker v_1 v_2 **else** $\mathsf{v}_1 = \mathsf{v}_1$ is immediately proven using the inequality. Instead, a prover that does not track inequalities needs to go through the following steps. First, it needs to take the branch eq t_1 $\mathsf{t}_2 = \top$. Second, it must rewrite t_1 into t_2 in le t_1 $\mathsf{t}_2 = \bot$, which

is present in the proof context. Using the reflexivity of le, the prover would derive the contradiction $\mathsf{le}\ t_1\ t_1 = \bot$, which eliminates this case. Besides the additional steps, this solution requires finding the useful equality to rewrite in which is not obvious, or incurs the cost of rewriting in all equalities.

In addition to reducing the steps in the proof, tracking inequalities requires to explore less alternatives (*breadth*) in a proof. Propel compactly represents a proof context as a flat conjunction of the form $(e_1 = e_2) \wedge \cdots \wedge (e_n \neq e_{n+1})$. A prover that does not track inequalities can express the same context by changing every inequality $e \neq K_0$ into $(e = K_1) \vee \cdots \vee (e = K_n)$ where K_i are the possible constructors of e's type. However, this solution introduces the need to consider all branches, one for each constructor, during the proof.

Proof search. Due to the exponential growth of the term space when considering all possible rewritings, we employ a strategy that limits the space to a number of "top" terms based on an arbitration order. We apply rewrites recursively to all subterms, resulting in sets of the "top"-most rewritten subterms. We reassemble the rewritten subterms and then continue to rewrite the assembled term itself. We perform this rewriting process on the complete syntax tree of the equation bottom-up, retaining a set of bounded size for the rewrite results of each subterm.

The selection of which trees to retain is guided by a heuristics that favors "simpler" trees. Algorithm 3 specifies how the heuristics scores different terms – terms with a higher score are considered simpler. The heuristics prioritizes data constructors (Line 1) over variables (Line 2), which in turn take precedence over pattern matches (Line 3), applications (Line 4) and abstractions (Line 5), the closer they are to the root of the tree. Despite its simplicity, our heuristic approach has proven effective in verifying algebraic properties of functions.

Algorithm 3 Syntax Tree Scoring

1: $(\!| \mathsf{K}\ \overline{e} |\!) = (4, \overline{(\!| e |\!)})$
2: $(\!| x |\!) = 3$
3: $(\!| e\ \mathsf{case}\ \{\overline{\mathsf{K} \Rightarrow e'}\} |\!) = (2, (\!| e |\!), \overline{(\!| e' |\!)})$
4: $(\!| e_1\ e_2 |\!) = (1, (\!| e_1 |\!), (\!| e_2 |\!))$
5: $(\!| \lambda\ x{:}T.e |\!) = (0, (\!| e |\!))$

The equality check is considered successful under two conditions: (1) if a contradiction arises within the equality set, indicating that the corresponding branch can never be reached or (2) if both sides of the equality are syntactically identical.

2.7 Limitations

A limitation of Propel lies in completing proofs that require unfolding definitions. Propel currently does not unfold definitions for performance reasons and instead relies solely on the discovered algebraic properties. Yet, sometimes unfolding is inevitable for three reasons: First, a needed algebraic property is not among Propel's predefined set of properties and not asserted explicitly by the developer. Second, a needed auxiliary property can only be generated after unfolding a constructor multiple times. Third, a property's proof may require unfolding some definitions where the definitions cancel each other out in order to complete the proof.

Like any any automated theorem prover for general-purpose languages, Propel's proof search cannot be both sound and complete. While Propel outperforms state-of-the-art competitors for algebraic properties, our evaluation shows some cases that Propel fails to prove although a proof exists, such as some properties of with bit vector multiplication, associativity of the LLWReg data type merge function and some monad laws (Section 4).

3 FORMALIZATION

This section introduces the core calculus for Propel, including the syntax (Section 3.1), the operational semantics (Section 3.2), and the type system (Section 3.3) with the proof rules for property annotations on functions (Section 3.4). The core calculus extends the simply-typed lambda calculus with the essential rules necessary to perform proofs of algebraic properties from type annotations.

3.1 Syntax

Propel's syntax extends the simply-typed lambda calculus with pattern matching, constructors, union types, a fixed point operator, and property annotations on functions.

Definition 1 (Syntax).

$$\begin{array}{lll} \text{Expressions} & \mathsf{e} ::= \mathsf{x} \mid \mathsf{e}_1\ \mathsf{e}_2 \mid \lambda^{\overline{\mathsf{r}}}\ \mathsf{x}{:}\mathsf{T}.\mathsf{e} \mid \mathsf{K} \mid \mathsf{e}\ \mathsf{case}\ \{\overline{\mathsf{K} \Rightarrow \mathsf{e}'}\} \mid \mathsf{fix}\ \mathsf{e} \\ \text{Values} & \mathsf{v} ::= \lambda^{\overline{\mathsf{r}}}\ \mathsf{x}{:}\mathsf{T}.\mathsf{e} \mid \mathsf{K}\ \overline{\mathsf{v}} \\ \text{Types} & \mathsf{T} ::= \mathsf{T}_1 \xrightarrow{\overline{\mathsf{r}}} \mathsf{T}_2 \mid \mathsf{K}\ \overline{+\ \mathsf{K}} \\ \text{Properties} & \mathsf{r} ::= \forall \overline{\mathsf{x} : \mathsf{T}}.\ \mathsf{e}_1 = \mathsf{e}_2 \mid \forall \overline{\mathsf{x} : \mathsf{T}}.\ \mathsf{e}_1 \neq \mathsf{e}_2 \end{array}$$

An expression is either a variable, an application, an abstraction annotated with a set of properties $\overline{\mathsf{r}}$, a constructor K, a pattern matcher that associates to each constructor a handler, and a fixed point operator that allows for recursion. To simplify the rules, patterns in a pattern matching expression do not bind fresh variables to each constructor's arguments as is the case with most languages. Instead, as we show in Section 3.2, a constructor's handler is expected to be a function that accepts that constructor's arguments. Moreover, we assume that patterns do not overlap, i.e., that no constructor is matched on more than once. A value is either an abstraction as usual or a constructor applied to one or more values. A type is either a function annotated with a set of properties $\overline{\mathsf{r}}$ or a union of one or more constructors. Thus constructors are akin to functions whose domain is the constructor's arguments and codomain is a type with the same name as the constructor's [24]. While the constructor expression and the constructor type are different we refer to both as constructors. Finally, a property is quantified equality between two expressions. To refer to the function annotated in the property, we assume that a fixed variable f is always used. For example, the identity function having the identity property is denoted $\mathsf{s}\ \lambda^{\forall \mathsf{x}:\mathsf{T}.\mathsf{f}\ \mathsf{x}=\mathsf{x}}\ \mathsf{x}{:}\mathsf{T}.\mathsf{x}$

Constructors. The calculus does not support defining new constructors. Instead, we assume that it is parametrized by a set of well-defined constructors which practically, in some implementation, is collected before type checking. This set is well-defined when it contains a finite set of constructors, each defined to take a finite number of arguments having a well-defined type in that set and returning itself. A well-defined type is either a finite union of constructors in a well-defined constructor set, or a function whose domain and co-domain are also well-defined. Moreover, constructors are expected to be inhabited by finite values only, for example the constructor K whose type is $\mathsf{K} \longrightarrow \mathsf{K}$ is not inhabited by finite values, and thus any constructor set that includes it is not well-defined. For example, the set of constructors $\{\mathsf{Z} : \mathsf{Z}, \mathsf{S} : (\mathsf{Z} + \mathsf{S}) \longrightarrow \mathsf{S}\}$ is well-defined.

Notation. We use $\overline{\mathsf{s}}$ to represent a sequence of zero or more s and $\overline{\mathsf{s}_\mathsf{i}}^{i}$ when the sequence is indexed by i. When it's not ambiguous we eliminate the superscript and use $\overline{\mathsf{s}_\mathsf{i}}$ instead. We use $\overline{\mathsf{s}}^{i,p(i)}$ to filter $\overline{\mathsf{s}}^{i}$ based on the index i with $p(i)$, a predicate that must hold true for all indexes i in $\overline{\mathsf{s}}^{i,p(i)}$. When grouping two forms sharing the same index, e.g., $\overline{\mathsf{x}_\mathsf{i}}$ and $\overline{\mathsf{T}_\mathsf{i}}$ into $\overline{\mathsf{x}_\mathsf{i} : \mathsf{T}_\mathsf{i}}$ we assume that the two lists have the same length and grouping is done pair-wise.

3.2 Evaluation Rules

We adopt a small-step call-by-value operational semantics $\mathsf{e}_1 \longrightarrow \mathsf{e}_2$ with evaluation context C. The notation $\mathsf{e}_1[\mathsf{x} \mapsto \mathsf{e}_2]$ refers to the capture-avoiding substitution of a variable x for expression e_2 in expression e_1. A substitution of a variable x must also be done inside properties r, on expressions and on types, when x is not bound with the $\forall$ quantifier.

Definition 2 (Evaluation Context). $C ::= [] \mid \mathsf{C}\ \mathsf{e} \mid \mathsf{v}\ \mathsf{C} \mid \mathsf{fix}\ \mathsf{C} \mid \mathsf{C}\ \mathsf{case}\ \{\overline{\mathsf{K} \Rightarrow \mathsf{e}}\}$

Definition 3 (Evaluation Rules).

$$\text{(E-App)}\ (\lambda^{\overline{r}}\, x : T.e_1)\ v \longrightarrow e_1[x \mapsto e_2] \qquad \text{(E-Case)}\ K_j\ \overline{v_j}\ \text{case}\ \{\overline{K_i \Rightarrow e_i}\} \longrightarrow e_j\ \overline{v_j}$$

$$\text{(E-Fix)}\ \text{fix}\ \lambda^{\overline{r}}\, x : T.e \longrightarrow e[x \mapsto \text{fix}\ \lambda^{\overline{r}}\, x : T.e] \qquad \text{(E-Context)}\ \frac{e \longrightarrow e'}{C[e] \longrightarrow C[e']}$$

E-App, E-Fix, and E-Context are standard. E-Case picks the handler of the constructor that matches the scrutinee's constructor and applies it to the constructor's arguments.

3.3 Type Checking

The typing context is defined in Definition 4.

Definition 4 (Typing Contexts).

Variable Context	$\Gamma ::= \overline{x : T}$	Constructor Context	$\Gamma_0 ::= \overline{K : T \xrightarrow{\text{inj}} K}$
Rule Context	$\mathcal{R} ::= \overline{r}$	Typing Context	$\mathbb{\Gamma} ::= \mathcal{R}; \Gamma_0; \Gamma$

Typing context Γ maps variables to types. Γ_0 denotes the typing context for constructors, mapping each constructor to a function that takes zero or more arguments and returns a value of the same type. Formally Γ_0 is the parameter defining the well-defined constructor set discussed earlier in this section. Importantly, constructors are expected to be injective, therefore the type of each constructor is expected to be an injective function which we denote with inj and define later. The rule context $\mathcal{R}$ stores all the known properties and rules. While $\mathcal{R}$ is not directly used by the type system they are fed back-and-forth between the typing rules and the proof rules introduced later.

The typing rules carry three contexts: the rule context $\mathcal{R}$, the constructor context Γ_0, and the variable context Γ, which we combine into a single context $\mathbb{\Gamma}$. We further write $\mathbb{\Gamma}, r$ for $\mathcal{R}, r$ and $\mathbb{\Gamma}, x : T$ for $\Gamma, x : T$ to extend the context, respectively.

The typing rules are presented in Definition 5

Definition 5 (Typing Rules).

$$\text{(T-Var)}\ \frac{x : T \in \mathbb{\Gamma}}{\mathbb{\Gamma} \vdash x : T} \qquad \text{(T-Cons)}\ \frac{K : T \in \mathbb{\Gamma}}{\mathbb{\Gamma} \vdash K : T} \qquad \text{(T-Fix)}\ \frac{\mathbb{\Gamma} \vdash e : T \xrightarrow{\overline{r}} T}{\mathbb{\Gamma} \vdash \text{fix}\ e : T}$$

$$\text{(T-App)}\ \frac{\mathbb{\Gamma} \vdash e_1 : T_1 \xrightarrow{\overline{r}} T_2 \qquad \mathbb{\Gamma} \vdash e_2 : T_1}{\mathbb{\Gamma} \vdash e_1\ e_2 : T_2} \qquad \text{(T-Sub)}\ \frac{\mathbb{\Gamma} \vdash e : T_1 \qquad T_1 \sqsubseteq T_2}{\mathbb{\Gamma} \vdash e : T_2}$$

$$\text{(T-Abs)}\ \frac{\mathbb{\Gamma}, x : T_1, \overline{r[f \mapsto t]} \vdash e : T_2 \qquad \overline{\mathbb{\Gamma} \vdash_{T_1 \to T_2} r} \qquad \overline{\overline{\mathbb{\Gamma}, \overline{r_j[f \mapsto t]}^{\,j,j<i}; \cdot \Vdash r_i[f \mapsto t]}^{\,i} \qquad t \stackrel{\text{def}}{=} \lambda^{\overline{r}}\, x : T_1.e}{\mathbb{\Gamma} \vdash \lambda^{\overline{r}}\, x : T_1.e : T_1 \xrightarrow{\overline{r}} T_2}$$

$$\text{(T-Aux)}\ \frac{\begin{array}{c} r \in aux(e) \\ \mathbb{\Gamma}; \cdot; \cdot \Vdash r \\ \mathbb{\Gamma}, r \vdash e : T \end{array}}{\mathbb{\Gamma} \vdash e : T} \qquad \text{(T-Case)}\ \frac{\mathbb{\Gamma} \vdash e : \overline{+\, K_i}^{\,i} \qquad \overline{\mathbb{\Gamma} \vdash K_i : \overline{T_{i,j}} \to^{j} K_i}^{\,i} \qquad \overline{\mathbb{\Gamma} \vdash e_i : \overline{T_{i,j}} \to^{j} T}^{\,i}}{\mathbb{\Gamma} \vdash e\ \text{case}\ \{\overline{K_i \Rightarrow e_i}\} : T}$$

T-Var, T-Cons, T-App, T-Sub, and T-Fix are standard. T-Sub uses the subtyping relation in Definition 6 to change the type of an expression into an equivalent or a weaker type. Propel's implementation uses this rule in three key places: (1) when type-checking function applications, Propel tries to check if a passed argument's type can be made to match the declared argument's type, (2) when type-checking a case expression it checks if the handled constructors form a permutation of the constructors in the union type of the scrutinee, and (3) in computing the least upper bound of the returned type of all case expression handlers. T-Case requires the scrutinee of a case expression to be a union type, that each constructor in the union has a corresponding handler, that each handler

is a function that takes the same arguments as its corresponding constructor, and that all handlers return the same type T which is the type of the case expression. T-Abs does not just type-check the function in the standard way, but also proves all the annotated properties. It works as follows: (1) it assumes all the properties hold when type-checking the function's body, (2) it checks that each annotated property is well-typed as defined in Definition 7, and (3) it goes in order over the annotated properties and proves each using all the known rules and the previously proven ones using the $\Phi \Vdash \mathsf{r}$ relation defined in Section 3.4. The order of properties is left undefined and any ordering may be chosen as it will not affect the soundness of the type system. Propel's implementation employs heuristics to guess the complexity of each property and orders the properties from least to most complex. If any property failed to be proven it is pushed again on the queue of properties to be proven. During the proof, the proof system can instantiate any property at recursive calls when arguments get smaller. This enables inductive reasoning as applying the property at recursive calls is equivalent to using the induction hypothesis. We assume the implementation checks that recursive calls happen on structurally smaller inputs which is ensured by termination checking. Finally, T-Aux uses the helper function *aux* that we leave undefined. We assume it computes a finite set of well-formed and well-typed conjectures that must be proven and that could be helpful in future proofs before type-checking continues. The calculus does not require any particular *aux* definition since, regardless which auxiliary properties are conjectured, *aux* can never impair the soundness of the prover: every auxiliary property needs to be proven before it can be used. We discuss the conjectures generated by the implementation in Sections 2.3 and 2.4. Auxiliary property generation may occur anytime, but in Propel's implementation it is attached to a lambda for pragmatic reasons: the auxiliary property generator can inspect the recursive function and conjecture useful auxiliary properties that aid the inductive proof.

Definition 6 defines the subtyping relation over types.

Definition 6 (Subtyping Relation).

$$\text{(S-Refl)}\ \mathsf{T} \sqsubseteq \mathsf{T} \qquad \text{(S-Union)}\ \mathsf{K}_1 \sqsubseteq \mathsf{K}_1 + \overline{\mathsf{K}_2} \qquad \text{(S-Perm)}\ \overline{\mathsf{K}_1} + \mathsf{K}_2 + \mathsf{K}_3 + \overline{\mathsf{K}_4} \sqsubseteq \overline{\mathsf{K}_1} + \mathsf{K}_3 + \mathsf{K}_2 + \overline{\mathsf{K}_4}$$

$$\text{(S-Func)}\ \frac{\mathsf{T_1}' \sqsubseteq \mathsf{T_1} \qquad \mathsf{T_2} \sqsubseteq \mathsf{T_2}'}{\mathsf{T_1} \xrightarrow{\bar{\mathsf{r}},\overline{\mathsf{r}'}} \mathsf{T_2} \sqsubseteq \mathsf{T_1}' \xrightarrow{\bar{\mathsf{r}}} \mathsf{T_2}'} \qquad \text{(S-Trans)}\ \frac{\mathsf{T_1} \sqsubseteq \mathsf{T_2} \qquad \mathsf{T_2} \sqsubseteq \mathsf{T_3}}{\mathsf{T_1} \sqsubseteq \mathsf{T_3}}$$

S-Refl and S-Trans are the usual subtyping rules that construct the reflexive and transitive closure, respectively, of the $\sqsubseteq$ relation. S-Union allows a constructor type to be ambiguated with other contructors in a union type. S-Perm allows the permutation of the constructor types in a union. And S-Func is the usual subtyping rule for functions with the additional constraints that the smaller function may extend the larger function's properties with more properties.

Definition 7 defines the well-typeness relation of a property.

Definition 7 (Well-Typed Property Relation).

$$\text{(W-Eq)}\ \frac{\mathbb{\Gamma}, \overline{\mathsf{x} : \mathsf{T_1}}, \mathsf{f} : \mathsf{T} \vdash \mathsf{e_1} : \mathsf{T_2} \qquad \mathbb{\Gamma}, \overline{\mathsf{x} : \mathsf{T_1}}, \mathsf{f} : \mathsf{T} \vdash \mathsf{e_2} : \mathsf{T_2}}{\mathbb{\Gamma} \vdash_{\mathsf{T}} \forall \overline{\mathsf{x} : \mathsf{T_1}}.\ \mathsf{e_1} = \mathsf{e_2}} \qquad \text{(W-Ineq)}\ \frac{\mathbb{\Gamma}, \overline{\mathsf{x} : \mathsf{T_1}}, \mathsf{f} : \mathsf{T} \vdash \mathsf{e_1} : \mathsf{T_2} \qquad \mathbb{\Gamma}, \overline{\mathsf{x} : \mathsf{T_1}}, \mathsf{f} : \mathsf{T} \vdash \mathsf{e_2} : \mathsf{T_2}}{\mathbb{\Gamma} \vdash_{\mathsf{T}} \forall \overline{\mathsf{x} : \mathsf{T_1}}.\ \mathsf{e_1} \neq \mathsf{e_2}}$$

The two rules defining a well-typed property guarantee that the two expressions being compared have a type in common. The type parameter to the $\vdash_{\mathsf{T}}$ relation is thus the type of the function to which the property is attached to and to which the symbol f must be mapped to.

Definition 8 defines the translation of commutativity, symmetry, associativity, reflexivity, idempotency, irreflexivity, selectivity, antisymmetry, transitivity, connectivity, and injectivity into quantified equalities that can be attached to functions and types.

Definition 8 (Interesting Properties). Propel's built-in properties are defined as follows:

$$\begin{aligned}
\text{comm} &:= \forall x_1 : T, x_2 : T.\ f\ x_1\ x_2 = f\ x_2\ x_1 & \text{sym} &:= \forall x_1 : T, x_2 : T.\ f\ x_1\ x_2 = f\ x_2\ x_1\\
\text{assoc} &:= \forall x_1 : T, x_2 : T, x_3 : T.\ f\ x_1\ (f\ x_2\ x_3) = f\ (f\ x_1\ x_2)\ x_3 & \text{refl} &:= \forall x : T.\ f\ x\ x = \top\\
\text{idem} &:= \forall x : T.\ f\ x\ x = x & \text{irefl} &:= \forall x : T.\ f\ x\ x = \bot
\end{aligned}$$

$$\begin{aligned}
\text{sel} &:= \forall x_1 : T, x_2 : T, \text{eq} : T \xrightarrow{\text{refl,sym,antisym}} T \to T.\ \text{eq}\ (f\ x_1\ x_2)\ x_1\ \text{case}\ \{\top \Rightarrow \top, \bot \Rightarrow \text{eq}\ (f\ x_1\ x_2)\ x_2\} = \top\\
\text{antisym} &:= \forall x_1 : T, x_2 : T, x_3 : T.\ f\ x_1\ x_2\ \text{case}\ \{\bot \Rightarrow x_3, \top \Rightarrow x_1\} = f\ x_2\ x_1\ \text{case}\ \{\bot \Rightarrow x_3, \top \Rightarrow x_2\}\\
\text{trans} &:= \forall x_1 : T, x_2 : T, x_3 : T.\ f\ x_1\ x_2\ \text{case}\ \{\bot \Rightarrow \top, \top \Rightarrow f\ x_2\ x_3\ \text{case}\ \{\bot \Rightarrow \top, \top \Rightarrow f\ x_1\ x_3\}\} = \top\\
\text{conn} &:= \forall x_1 : T, x_2 : T, x_3 : T.\ f\ x_1\ x_2\ \text{case}\ \{\top \Rightarrow x_3, \bot \Rightarrow f\ x_2\ x_1\ \text{case}\ \{\top \Rightarrow x_3, \bot \Rightarrow x_1\}\}\\
&\qquad = f\ x_1\ x_2\ \text{case}\ \{\top \Rightarrow x_3, \bot \Rightarrow f\ x_2\ x_1\ \text{case}\ \{\top \Rightarrow x_3, \bot \Rightarrow x_2\}\}\\
\text{inj} &:= \forall x_1 : T, x_2 : T, \text{eq} : T \xrightarrow{\text{refl,sym,antisym}} T \to T.\ \text{eq}\ (f\ x_1)\ (f\ x_2)\ \text{case}\ \{\bot \Rightarrow \top, \top \Rightarrow \text{eq}\ x_1\ x_2\} = \top
\end{aligned}$$

3.4 Property Checking

This section presents the rules of the property prover. Recall that the prover is used by the typing rules T-Abs and T-Aux.

Definition 9 (Proof Contexts).

Proof Context	$\Phi ::= \Gamma; \mathcal{E}^+; \mathcal{E}^-$	Equalities	$\mathcal{E}^+ ::= \overline{e_1 = e_2}$
Inequalities	$\mathcal{E}^- ::= \overline{e_1 \neq e_2}$	(In)equality	$Q^0 ::= e_1 = e_2$
(In)equality Ctx.	$Q^1 ::= [] = e \mid e = [] \mid [] \neq e \mid e \neq []$		$\mid e_1 \neq e_2$
Dual (In)equalities Ctx.	$Q^2 ::= [] = Q^1 \mid Q^1 = [] \mid [] \neq Q^1 \mid Q^1 \neq []$	Optional	$Q^{1?} ::= Q^0 \mid Q^1$

The proof context Φ is defined exactly as Γ with the two additional contexts storing unquantified equalities and inequalities. To append rules or variable bindings we do so directly on Φ unambiguously. Similarly to the context C from Definition 2, we define the single and dual (in)equality contexts as Q^1 and Q^2 respectively. We write $Q^1[e]$ to mean the (in)equality with e replacing the single hole [] in Q^1. We write $Q^2[e_1, e_2]$ to mean the (in)equality with e_1 replacing the first hole and e_2 replacing the second.

Proof rules. Proof rules of the form $\Phi \Vdash r$ define the relationship between the proof goal, the proof strategy, and the known (in)equalities, and typing information in the proof context.

Definition 10 (Proof Rules).

$$\text{(P-Intro)}\ \frac{\Phi, \overline{x : T} \Vdash Q^0}{\Phi \Vdash \forall \overline{x : T}.Q^0} \qquad \text{(P-Contra}^-\text{)}\ \frac{e \neq e \in \Phi}{\Phi \Vdash r} \qquad \text{(P-Hyp)}\ \frac{Q^0 \in \Phi}{\Phi \Vdash Q^0}$$

$$\text{(P-Contra}^\cup\text{)}\ \frac{K\ \overline{e_i'} = e \in \Phi \qquad e \in e_i}{\Phi \Vdash r} \qquad \text{(P-Abs)}\ \frac{\Gamma \vdash e : T \qquad \Phi[e \mapsto x], x : T, x = e \Vdash r[e \mapsto x]}{\Phi \Vdash r}$$

$$\text{(P-Eq)}\ \frac{e_1 = e_2 \in \Phi \qquad \Phi, Q_1^{1?}[e_2] \Vdash Q_2^{1?}[e_2]}{\Phi, Q_1^{1?}[e_1] \Vdash Q_2^{1?}[e_1]} \qquad \text{(P-Derive)}\ \frac{\Phi \triangleright \Phi' \qquad \Phi' \Vdash r}{\Phi \Vdash r}$$

$$\text{(P-Case)}\ \frac{\overline{\Phi, \overline{x_{i,j} : T_{i,j}}^{j}, e = K_i\ \overline{x_{i,j}}^{j}, \overline{e \neq K_n\ \overline{x_{n,j}}^{j}}^{n, n \neq i} \Vdash r}^{i} \qquad \Gamma \vdash e : \overline{+ K_i}^{i} \qquad \overline{\Gamma \vdash K_i : \overline{T_{i,j}} \to^{j} K_i}^{i}}{\Phi \Vdash r}$$

P-Intro introduces all the quantified variables in an (in)equality in the context before proving the unquantified version. P-Contra$^-$ allows to terminate any proof if two syntactically equal expressions are unequal in the proof context encoding *reductio ad absurdum*. P-Hyp allows to conclude a proof of an (in)equality if it occurs in the proof context. P-Contra$^\cup$ is another proof by contradiction

strategy relying on the fact that recursive equalities applied to constructors result in infinite values which are not possible to construct in the calculus. P-Abs allows to replace all occurrences of some expression modulo variable renaming with a fresh variable everywhere in the proof context and the goal. The notation $[e \mapsto x]$ is used to represent this substitution. The notation $\Phi[e \mapsto x]$ denotes the substitution in the (in)equalities in the proof context, i.e., $\Gamma; \mathcal{E}^+[e \mapsto x]; \mathcal{E}^-[e \mapsto x]$. Crucially e must type under Γ to guarantee that it does not include free-variables once it's lifted from its surrounding context in a larger expression. P-Eq allows to make use of a known equality to replace one side of the equality with another somewhere in the context and/or in the goal. P-Derive allows the context to derive new (in)equalities which may be useful in the proof. The relation $\Phi \triangleright \Phi'$ that derives new information is defined in Definition 11. Finally, P-Case performs case analysis on a union-typed expression: (1) it checks that the analyzed expression has a union type, (2) it introduces a fresh variable for each argument of each constructor, (3) for each constructor, it forks the proof with the equality that the expression is equal to some construction of said constructor and thus unequal to all other constructions using the other constructors.

The rules P-Hyp, P-Abs, and P-Eq are sufficient to prove that equality as represented is symmetric and transitive. Reflexivity is left to D-Refl.

Derivation rules. The derivation rules in Definition 11 define how known properties, (in)equalities, and typing information can be used to derive new (in)equalities.

Definition 11 (Derivation Rules).

$$\text{(D-Refl)}\ \frac{\Gamma \vdash e : T}{\Phi \triangleright \Phi, e = e} \qquad \text{(D-Inst)}\ \frac{\forall \overline{x : T}.Q^0 \in \Phi \qquad \overline{\Gamma \vdash x\sigma : T}}{\Phi \triangleright \Phi, Q^0\sigma}$$

$$\text{(D-Ineq)}\ \frac{\Gamma \vdash e_1 : T_1 \qquad \Gamma \vdash e_2 : T_2 \qquad T_1 \not\subseteq T_2 \qquad T_2 \not\subseteq T_1}{\Phi \triangleright \Phi, e_1 \neq e_2} \qquad \text{(D-Arg)}\ \frac{e_1\ \overline{e_2}\ e_3\ \overline{e_4} \neq e_1\ \overline{e_2}\ e_3'\ \overline{e_4} \in \Phi}{\Phi \triangleright \Phi, e_3 \neq e_3'}$$

$$\text{(D-Rule)}\ \frac{\Gamma \vdash e : T_1 \xrightarrow{\bar{r}} T_2}{\Phi \triangleright \Phi, \overline{r[f \mapsto e]}} \qquad \text{(D-App)}\ \frac{Q^1[(\lambda^{\bar{r}}\, x{:}T.e_1)\ e_2] \in \Phi}{\Phi \triangleright \Phi, Q^1[e_1[x \mapsto e_2]]}$$

$$\text{(D-Case)}\ \frac{Q^1[K_i\ \overline{e_i'}\ \text{case}\ \{\overline{K_j \Rightarrow e_j}\}] \in \Phi}{\Phi \triangleright \Phi, Q^1[e_i\ \overline{e_i'}]} \qquad \text{(D-Fun)}\ \frac{Q^2[\lambda^{\bar{r}}\, x{:}T.e_1,\ e_2] \in \Phi}{\Phi \triangleright \Phi, x : T, Q^2[e_1,\ e_2\ x]}$$

D-Refl allows the context to derive that any well-typed expression is equal to itself. With this rule, it is possible to prove that equality is an equivalence relation. D-Inst allows a quantified property to be instantiated to particular values given a mapping σ of the quantified variables. Propel's implementation makes use of this rule whenever a property can be unified with a subexpression in the proof goal. D-Ineq allows to derive that two expressions whose types do not have a least upper bound must be inequal. This rule allows us to derive that values constructed with different constructors must be unequal. D-Arg relies on the fact that the language semantics are deterministic and thus if two copies of a function application are unequal but have all their arguments but one equal then that argument must be the one separating both application and both are thus different. D-Rule allows to move a property from type annotation to the rule context. D-App and D-Case allow the derivation engine to do symbolic evaluation on the expressions. While the calculus adds the new rewritten expression alongside the old one in the proof context, Propel's implementation discards the old expression to save memory and tame the proof runtime. Finally D-Fun encodes functional extensionality and says that two functions are equal when their bodies are equal.

3.4.1 Soundness. The calculus is type safe as it satisfies the progress and preservation theorems.

Theorem 1 (Progress). *If* $\Gamma_0; \cdot \vdash e : T$ *then* e *is a value or there exists* e' *such that* $e \longrightarrow e'$.

Theorem 2 (Preservation). *If* $\Gamma_0; \cdot \vdash e : T$ *and* $e \longrightarrow e'$ *then* $\cdot; \Gamma_0; \cdot \vdash e' : T$.

We further prove the soundness property that, if an expression is given a function type annotated with a property then the property truly holds. For example, if an expression e computes a commutative function, i.e., $\Gamma_0; \cdot \vdash T \xrightarrow{\text{comm}} T \to T$ then $e\ x_1\ x_2 = e\ x_2\ x_1$ for every x_1, x_2.

To this end, we first define equality. The first option is syntactic equality which indicate $=$, is too weak. For example, it does not allow us to express idempotence of the logical `or` function: $\forall x : 2.x$ case $\{\top \Rightarrow \top, \bot \Rightarrow x\} = x$. Clearly, both terms are not syntactically equal. Yet, they do normalize to the same expression. A more suitable definition of equality is syntactic equality up to normal forms. Definition 12 presents a definition that conveys our intuition for equality for a type T, which we symbolize with $\equiv_T$. It states that two expressions of (1) a constructor type are equal when they reduce to a construction of that constructor and its arguments are pair-wise equal, (2) a union type are equal when they are equal at a particular type in that union, (3) a function type when they normalize to equal values when applied to equal inputs.

Definition 12 (Equality up to normal forms). Given a data type K and two expressions e_1, e_2

- $e_1 \equiv_K e_2$ if and only if both have the same type K, i.e., $\Gamma_0; \cdot \vdash e_1 : K$ and $\Gamma_0; \cdot \vdash e_2 : K$, both reduce to constructions of that constructor, i.e., $e_1 \to^* K\ \overline{v_1}$ and $e_2 \to^* K\ \overline{v_2}$, the constructor takes as many arguments as provided, i.e., $\Gamma_0; \cdot \vdash K : \overline{T} \to K$, and each argument is pairwise equivalent at its expected type, i.e., $\overline{v_1 \equiv_T v_2}$.
- $e_1 \equiv_{\overline{+ K_i}} e_2$ if and only if there exists an i such that $e_1 \equiv_{K_i} e_2$.
- let $T = T_1 \xrightarrow{\bar{r}} T_2$ then $e_1 \equiv_T e_2$ if and only if both have the same type T, i.e., $\Gamma_0; \cdot \vdash e_1 : T$, and $\Gamma_0; \cdot \vdash e_2 : T$, and assuming $e_1 \to^* \lambda^{\bar{r}}\, x : T_1.e_{11}$ and $e_2 \to^* \lambda^{\bar{r}}\, x : T_1.e_{22}$ then for any $e_3, e_4, {T_1}'$ such that $e_3 \equiv_{{T_1}'} e_4$ and ${T_1}' \sqsubseteq T_1$ then $e_{11}[x \mapsto e_3] \equiv_{T_2} e_{22}[x \mapsto e_4]$.

We may leave out the type annotation from $\equiv_T$ when it is clear from the context.

To show that Definition 12 matches our intuition of equality up to normal forms, we prove a fortiori that $\equiv$ is preserved across $\longrightarrow$ and that $\equiv$ is an equivalence relation.

Lemma 1. *Assume* $e_1 \longrightarrow {e_1}'$ *then* $e_1 \equiv_T e_2$ *if and only if* ${e_1}' \equiv_T e_2$.

Lemma 2. $\equiv_T$ *is an equivalence relation at well-typed expressions.*

Definition 13 provides a well-typed substitution $\sigma \vDash \mathbb{\Gamma}$ when every variable in $\mathbb{\Gamma}$ is mapped to an expression of the same type under the empty context. Definition 14 defines equivalent substitutions under $\mathbb{\Gamma}$ to be the ones that map every variable to equal expressions. In Lemma 3, we show that expressions equivalent under the same substitution are equivalent under equivalent substitutions.

Definition 13. Let $\mathbb{\Gamma}$ and σ be given. $\sigma \vDash \mathbb{\Gamma}$ when $\cdot; \Gamma_0; \cdot \vdash \sigma(x) : T$ for every $x : T \in \mathbb{\Gamma}$.

Definition 14. Let $\mathbb{\Gamma}, \sigma_1 \vDash \mathbb{\Gamma}$, and $\sigma_2 \vDash \mathbb{\Gamma}$ be given. $\sigma_1 \equiv \sigma_2$ when $x\sigma_1 \equiv_T x\sigma_2$ for every $x : T \in \mathbb{\Gamma}$.

Lemma 3. *Let* $\mathbb{\Gamma} \vdash e_1 : T$ *and* $\mathbb{\Gamma} \vdash e_2 : T$ *and* $\sigma_1 \equiv \sigma_2$. *If* $e_1\sigma_1 \equiv e_2\sigma_1$ *then* $e_1\sigma_1 \equiv e_2\sigma_2$.

Armed with these lemmas we define a valid proof context as one where all its equalities and properties are equal up to normal form, and inequalities are provably unequal up to normal form which we denote with $\not\equiv_T$.

Definition 15. Φ is said to be valid when a substitution $\sigma \vDash \mathbb{\Gamma}$, dubbed the witness, exists such that $\mathcal{E}^+_{\equiv}\sigma$ and $\mathcal{E}^-_{\equiv}\sigma$ are true under the assumption that $\mathcal{R}_{\equiv}\sigma$. We define $\mathcal{E}^+_{\equiv}$, $\mathcal{E}^-_{\equiv}$, and $\mathcal{R}_{\equiv}$ as the interpretation of $\mathcal{E}^+, \mathcal{E}^-$ and $\mathcal{R}$, respectively, as follows: if $\mathcal{E}^+ = \overline{e_1 = e_2}$ then $\mathcal{E}^+_{\equiv} = \overline{e_1 \equiv e_2}$, if $\mathcal{E}^- = \overline{e_1 \neq e_2}$ then $\mathcal{E}^-_{\equiv} = \overline{e_1 \not\equiv e_2}$, if $r = \overline{\forall x : T.e_1 = e_2}$ then $r_{\equiv} = \overline{\lambda x : T.e_1 \equiv \lambda x : T.e_2}$, if $r = \overline{\forall x : T.e_1 \neq e_2}$ then $r_{\equiv} = \overline{\lambda x : T.e_1 \not\equiv \lambda x : T.e_2}$, if $\mathcal{R} = \bar{r}$ then $\mathcal{R}_{\equiv} = \overline{r_{\equiv}}$, where the many denotes conjunction.

Next, we prove the followng soundness theorems: a proof context derived from a valid one is itself valid (Theorem 3), every rule that is proved in a valid context holds (Theorem 4) , and if a function can be typed then its properties hold (Theorem 5). Finally, we prove the main soundness theorem (Theorem 6): annotated properties hold on any expression.

Theorem 3. *If* Φ *is valid and* $\Phi \triangleright \Phi'$*, then* Φ' *is valid.*

Theorem 4. *If* Φ *is valid and* σ *is its witness and* $\Phi \Vdash \mathsf{r}$*, then* $(\mathsf{r}\sigma)_{\equiv}$.

Theorem 5. *If* $\Gamma \vdash \lambda^{\bar{\mathsf{r}}}\ \mathsf{x}:\mathsf{T}.\mathsf{e}$ *and* $\sigma \vDash \Gamma$ *and* $\mathcal{R}_{\equiv}\sigma$*, then* $\overline{\mathsf{r}_{\mathsf{T}_1}[\mathsf{f} \mapsto \lambda^{\bar{\mathsf{r}}}\ \mathsf{x}:\mathsf{T}.\mathsf{e}\sigma]}_{\equiv}$.

Theorem 6. *If* $\Gamma_0; \cdot \vdash \mathsf{e} : \mathsf{T}_1 \xrightarrow{\bar{\mathsf{r}}} \mathsf{T}_2$ *and* e *normalizes, then* $\overline{\mathsf{r}_{\mathsf{T}_1}[\mathsf{f} \mapsto \mathsf{e}]}_{\equiv}$.

3.4.2 Equality Lifting and Embedding. Selection and connectedness in Section 2.3 require a disjunction of equalities, yet as presented, the equality set $\mathcal{E}^+$ is a conjunction of equalities. By embedding disjunction and equality from the meta- to the expression-level we can overcome this limitation. Luckily, equality is the unique relation that is reflexive, symmetric, and antisymmetric, thus embedding equality is akin to using a function with these properties. This embedding is expressed through the quantification over a reflexive, symmetric, and antisymmetric function in Definition 8.

However, embedding meta-level (in)equalities is only useful if they can be lifted back later. The following two theorems prove that both lifting and embedding can occur at any time.

Theorem 7. *Given a proof context* Φ *with* $\mathsf{eq} : \mathsf{T} \xrightarrow{\text{refl,sym,antisym}} \mathsf{T} \to \mathsf{T} \in \Phi$ *and* $\mathsf{eq}\ \mathsf{a}\ \mathsf{b} = \bot \in \Phi$ *there exists a sequence of derivation* $\Phi \triangleright^n \Phi'$ *such that* $\mathsf{a} \neq \mathsf{b} \in \Phi'$.

Theorem 8. *Given a proof context* Φ *with* $\mathsf{eq} : \mathsf{T} \xrightarrow{\text{refl,sym,antisym}} \mathsf{T} \to \mathsf{T} \in \Phi$ *and* $\mathsf{eq}\ \mathsf{a}\ \mathsf{b} = \top \in \Phi$ *there exists a sequence of derivation* $\Phi \triangleright^n \Phi'$ *such that* $\mathsf{a} = \mathsf{b} \in \Phi'$.

4 EVALUATION AND IMPLEMENTATION

Propel is implemented in Scala 3. It can verify a core language in S-expression syntax and Scala code through its DSL. Its core is about 10 K lines long spread throughout 46 files. The binary is compiled to native machine code using LLVM.

To evaluate our approach, we selected case studies from a variety of domains where algebraic properties, such as commutativity or associativity, are essential for correct execution. We compare against five state-of-the-art verifiers in five different domains, checking a total of 142 properties. We believe their importance in many domains justifies a proof technique that specializes on these properties and enables proving them with a high degree of automation. Instead, existing automated theorem provers are more general.

4.1 Case Studies

We investigated the following areas to cover a variety of use cases across different domains. Thereby, we focus on the set of essential algebraic properties explored in this paper.

Numeric operations. Segment 1 of Table 1 investigates common numeric operations – such as addition, multiplication, maximum, minimum – in isolation. Note that the following more complex use cases often rely on a combination of such operations. We explore two ways of modeling numerals: (1) the usual Peano encoding of natural numbers and (2) a bit vector encoding closer to the hardware bit representation. For (1), we provide the results for the different variations of addition from Section 2.

In addition to the basic algebraic properties that Propel tracks in the types and that concern single functions, we also prove left and right identities of the operations. While the emphasis of Propel's design is not properties on the combination of multiple functions, the properties which Propel can reason about are instrumental in proving such combining properties.

Replicated Data Types. Segment 2 investigates widely-used conflict-free replicated data types (CRDTs) [41]. CRDTs require a merge function that is commutative, associative, and idempotent, i.e., a join on a semilattice, to guarantee merging states without conflicts. We compare versions using Peano numbers and bit vectors.

Data Flow Engines. Segment 3 investigates data flow engines which use associative operations to parallelize data processing. For example, a reduce or a fold operation can run in parallel on different nodes, if the operation is associative as the result on different data can be composed. Existing engines trust the annotation provided by the developer that a function is associative and this property is not checked by the framework. We selected both widely-used and recent open-source engines: Naiad [30], Timely [2] Noir [28], and Apache Beam [1]. We verified the associativity of several functions used in the respective engine's unit tests or usage examples. For example, when merging the results, the WordCount unit test performs addition which must be associative.

Divide and Conquer. Segment 4 investigates well-known text book divide-and-conquer algorithms. Such algorithms require associative operations in their conquer phase to combine the results that were independently computed in the divide phase. We examined the Quicksort algorithm and parallel-friendly implementations of the numeric maximum and minimum over binary trees.

Type Class Laws. Segment 5 investigates some properties beyond the usual algebraic properties, which Propel tracks in the types, such as commutativity and associativity. Although not the main focus of our approach, we demonstrate how the algebraic reasoning of Propel can also help prove type class laws such as composition laws or left and right identities. For our investigation, we examined the type class instances defined in Haskell's Prelude [33].

4.2 TIP benchmarks

Segment 6 applies Propel to the subset of the TIP benchmarks [16], designed for automated inductive theorem provers, that checks the fundamental algebraic properties investigated in this paper, namely those that check commutativity, selection, idempotence, injectivity, associativity, reflexivity, irreflexivity, antisymmetry, symmetry, connectedness or transitivity.

4.3 Comparison and Parameter Analysis

We consider the following state-of-the-art inductive theorem provers and SMT solvers that can reason by induction. **HipSpec** [15] adopts a bottom-up approach (*theory exploration*), constructing a library of lemmas before attempting a proof. **Zeno** [44], on the other hand, employs *lemma discovery through generalization*, synthesizing and proving lemmas on-demand. **CycleQ** [23], an extension of the GHC Haskell compiler, leverages cyclic proof theory [10, 45]. All three tools parse

Table 1. Evaluation Results.

Domain	Function	Property	HipSpec	Zeno	CycleQ	cvc5	Vampire	Propel (no ≠)	Propel (aux #)	Propel
Segment 1.	*Numeric Operations.*									
Peano	Addition (add_1)	comm	1	0	⧗	0	0	1	2	0
		assoc	5	0	1	0	0	1	2	0
		left id	1	0	1	0	0	1	0	0
		right id	1	0	1	0	0	1	0	0
	Addition (add_2)	comm	1	0	⧗	0	⧗	1	6	0
		assoc	3	✗	⧗	5	⧗	1	6	0
		left id	1	0	1	0	0	1	0	0
		right id	1	0	1	0	0	1	0	0
	Addition (add_3)	comm	13	0	1	0	⧗	0	0	0
		assoc	36	✗	⧗	⧗	⧗	0	2	0
		left id	10	0	1	0	0	0	0	0
		right id	6	0	1	0	0	0	0	0
	Multiplication	comm	2	0	⧗	⧗	⧗	⧗	8	0
		assoc	13	0	⧗	⧗	⧗	⧗	12	0
		left id	2	0	1	⧗	⧗	⧗	1	0
		right id	2	0	1	⧗	⧗	⧗	2	0
	Minimum	comm	1	0	1	0	⧗	0	0	0
		assoc	5	0	1	0	⧗	0	0	0
		idem	1	0	1	0	0	0	0	0
	Maximum	comm	1	0	1	0	⧗	0	0	0
		assoc	5	0	1	0	⧗	0	0	0
		idem	1	0	1	0	0	0	0	0
		left id	1	0	1	0	0	0	0	0
		right id	1	0	1	0	0	0	0	0
Bit vectors	Addition	comm	⧗	0	1	0	⧗	0	0	0
		assoc	⧗	✗	⧗	⧗	⧗	0	10	0
		left id	⧗	0	1	0	0	0	0	0
		right id	⧗	0	1	0	0	0	0	0
	Multiplication	comm	⧗	⧗	⧗	⧗	⧗	✗	–	✗
		assoc	⧗	⧗	⧗	⧗	⧗	✗	–	✗
		left id	⧗	0	1	0	0	✗	–	✗
		right id	⧗	0	1	⧗	⧗	2	8	2
	Addition (mod 2^n)	comm	⧗	0	1	0	⧗	0	0	0
		assoc	⧗	✗	⧗	⧗	⧗	0	20	0
	Multiplication (mod 2^n)	comm	⧗	✗	⧗	⧗	⧗	2	20	2
		assoc	⧗	✗	⧗	⧗	⧗	✗	–	✗
	Minimum	comm	⧗	✗	⧗	0	⧗	6	0	3
		assoc	⧗	✗	⧗	⧗	⧗	✗	0	3
		idem	⧗	0	1	0	0	4	0	3
	Maximum	comm	⧗	⧗	⧗	0	⧗	6	0	3
		assoc	⧗	⧗	⧗	⧗	⧗	✗	0	3
		idem	⧗	0	1	0	0	4	0	3
		left id	⧗	0	1	0	0	4	0	3
		right id	⧗	0	1	0	0	4	0	3

Haskell code and attempt to prove the properties defined in it. We also compare against two solvers capable of inductive reasoning: **cvc5** [6] and **Vampire** [22].

To ensure comparability between the automated theorem provers and SMT solvers, we reimplemented the relevant data types with inductive definitions (e.g., Peano or bit vector encodings for natural numbers). Hence, the SMT solvers do not exploit some of their theories, e.g., for numbers, arrays, or sets, but still use some theories, in particular for data types. This approach is in line with common benchmarks for theorem provers, to set up a meaningful comparison between different provers. Aiming at full automation for the algebraic properties explored in this paper, we only specify the final property the developer is interested in and no intermediate auxiliary properties, leaving the entire verification process to the prover. As the provers require a specific import format (like SMTLIB2 or different variants of Haskell), we reimplemented each function for every prover, striving for implementations that are as similar as possible in Haskell (for HipSpec, Zeno and CycleQ), in SMTLIB2 (for cvc5 and Vampire) and in Propel.

We executed a new instance of the prover for every file on a Intel Core i7-1185G7, 3 GHz, 32 GiB setup. We set a timeout of one minute for every proof, as provers might diverge in the proof search. We consider this a reasonable duration for interactive theorem proving. All provers are in native binaries, hence no VM startup time is required. We further conducted a parameter analysis by enabling or disabling the tracking of inequalities and varying the threshold for the number of generated auxiliary properties to investigate which aspects of our approach contribute to Propel's advantage over state of the art verifiers.

4.4 Results

In Table 1, a ✗ indicates that the prover terminated unsuccessfully within the time limit. A ⧗ indicates that the prover timed out. Otherwise, we report the verification time in seconds. The last column shows the results for Propel. The preceding columns show the results of Propel without tracking inequalities ("no ≠" column) and the number of auxiliary properties explored by Propel for the proof ("aux #" column).

The results demonstrate Propel's deductive power to prove fundamental algebraic properties in many

Table 1. Evaluation Results (continued).

Domain	Function	Property	HipSpec	Zeno	CycleQ	cvc5	Vampire	Propel (no ≠)	Propel (aux #)	Propel
Segment 2.	*Replicated Data Types.*									
	GSet	comm	1	✗	1	–	–	0	0	0
		assoc	1	✗	1	–	–	0	0	0
		idem	1	✗	1	–	–	0	0	0
Peano	GCounter	comm	1	0	1	⧗	⧗	⧗	0	1
		assoc	1	0	1	⧗	⧗	⧗	0	1
		idem	1	0	1	3	0	⧗	0	1
	LWWReg	comm	55	✗	1	⧗	⧗	2	0	1
		assoc	⧗	✗	1	⧗	⧗	2	0	1
		idem	18	0	1	5	0	2	0	1
Bit vectors	GCounter	comm	⧗	⧗	1	⧗	⧗	⧗	0	5
		assoc	⧗	⧗	1	⧗	⧗	⧗	0	5
		idem	⧗	0	1	⧗	1	⧗	0	5
	LWWReg	comm	⧗	⧗	1	⧗	⧗	✗	0	4
		assoc	⧗	✗	1	⧗	⧗	✗	–	✗
		idem	⧗	0	1	⧗	⧗	12	0	4
Segment 3.	*Data Flow Engines.*									
Timely	Word Count	assoc	5	0	1	0	0	1	2	0
Noir	Word Count	assoc	5	0	1	0	0	1	2	0
Naiad	Word Count	assoc	5	0	1	0	0	1	2	0
Beam	min	comm	1	0	1	0	⧗	0	0	0
		assoc	5	0	1	0	⧗	0	0	0
		idem	1	0	1	0	0	0	0	0
	max	comm	1	0	1	0	⧗	0	0	0
		assoc	5	0	1	0	⧗	0	0	0
		idem	1	0	1	0	0	0	0	0
	sum	comm	1	0	⧗	0	0	1	2	0
		assoc	5	0	1	0	0	1	2	0
	times	comm	2	0	⧗	⧗	⧗	0	8	0
		assoc	13	0	⧗	⧗	⧗	0	12	0
Segment 4.	*Divide and Conquer.*									
Quicksort	merge	assoc	43	0	1	0	0	3	6	0
Tree	min	comm	1	0	1	0	⧗	0	0	0
		assoc	5	0	1	0	⧗	0	0	0
		idem	1	0	1	0	0	0	0	0
Tree	max	comm	1	0	1	0	⧗	0	0	0
		assoc	5	0	1	0	⧗	0	0	0
		idem	1	0	1	0	0	0	0	0
Segment 5.	*Type Class Laws.*									
Functor	Maybe map	id	1	0	1	–	–	0	0	0
		comp	1	0	1	–	–	0	0	0
	List map	id	2	0	1	–	–	1	0	0
		comp	2	0	1	–	–	0	0	0
	Function map	id	12	✗	1	–	–	0	0	0
		comp	12	✗	1	–	–	0	0	0
	Pairs map	id	1	0	1	–	–	0	0	0
		comp	1	0	1	–	–	0	0	0
	State map	id	1	✗	✗	–	–	0	0	0
		com	1	✗	✗	–	–	0	0	0
Semigroup	Maybe op	assoc	1	0	1	–	–	1	2	0
	List op	assoc	1	0	1	–	–	3	10	0
	Function op	assoc	36	✗	1	–	–	1	2	0
	Pair op	assoc	1	0	1	–	–	⧗	2	6
	State op	assoc	⧗	✗	✗	–	–	3	2	0
Monad	Maybe bind and return	assoc	1	✗	1	–	–	0	0	0
		left id	1	0	1	–	–	0	0	0
		right id	1	0	1	–	–	0	0	0
	List bind and return	assoc	9	✗	⧗	–	–	✗	–	✗
		left id	3	✗	✗	–	–	3	1	0
		right id	2	0	1	–	–	3	10	0
	Function bind and return	assoc	1	✗	1	–	–	0	0	0
		left id	5	✗	1	–	–	✗	–	✗
		right id	5	✗	1	–	–	✗	–	✗
	Pair bind and return	assoc	⧗	✗	✗	–	–	2	2	0
		left id	⧗	✗	✗	–	–	1	2	0
		right id	⧗	0	1	–	–	1	1	0
	State bind and return	assoc	1	✗	✗	–	–	0	0	0
		left id	8	✗	✗	–	–	✗	–	✗
		right id	1	✗	✗	–	–	✗	–	✗

domains. Propel's design was influenced by investigating the kind of properties required by replicated data types. We applied Propel to the other domains only after the implementation of Propel was completed, i.e., they did not influence its design. The state-of-the-art provers do not perform as well as Propel in terms of the number of properties they can prove, especially when applied to more realistic use cases such as CRDTs, divide-and-conquer algorithms, data flow engines, and type class laws. We attribute this to Propel's focus on algebraic properties, specifically generating auxiliary properties that proved useful for such properties, and tracking both equalities and inequalities. The SMT solvers we used are not able to reason about higher-order functions. Hence, they could not verify the GSet CRDT, which represents sets as functions, and the type class laws. Hence, we leave the corresponding entries in the table blank ("–").

Table 1. Evaluation Results (continued).

Function	Property	HipSpec	Zeno	CycleQ	cvc5	Vampire	Propel (no ≠)	Propel (aux #)	Propel
Segment 6. TIP Benchmarks for Algebraic Properties.									
bin_plus	comm	17	0	1	0	⧖	1	0	1
	assoc	44	✗	⧖	⧖	⧖	1	21	1
bin_times	comm	48	⧖	⧖	⧖	⧖	2	21	2
	assoc	⧖	⧖	⧖	⧖	⧖	✗	–	✗
int_plus	comm	⧖	0	⧖	0	0	⧖	–	⧖
	assoc	⧖	✗	⧖	0	0	⧖	–	⧖
int_times	comm	⧖	✗	⧖	⧖	⧖	⧖	8	1
	assoc	⧖	✗	⧖	⧖	⧖	⧖	–	✗
list_append	assoc	43	0	1	0	0	3	6	0
nat_geq	antisym	8	0	⧖	0	⧖	✗	0	0
	refl	3	0	1	0	0	0	0	0
	trans	12	✗	⧖	0	⧖	0	0	0
nat_gt	asym	4	1	⧖	0	⧖	0	0	0
	irefl	0	0	1	0	⧖	0	0	0
	trans	12	✗	⧖	0	⧖	0	0	0
nat_leq	antisym	1	0	⧖	0	⧖	✗	0	0
	refl	0	0	1	0	0	0	0	0
	trans	0	0	⧖	0	⧖	0	0	0
nat_lt	asym	1	1	⧖	0	59	0	0	0
	irefl	0	0	1	0	⧖	0	0	0
	trans	7	✗	⧖	0	⧖	0	0	0
nat_min	comm	0	✗	⧖	0	⧖	✗	0	0
	assoc	4	✗	⧖	0	⧖	0	0	0
	idem	0	0	⧖	0	0	0	0	0
nat_max	comm	6	✗	⧖	0	⧖	✗	0	0
	assoc	8	✗	⧖	0	⧖	0	0	0
	idem	0	0	⧖	0	⧖	0	0	0
nat_plus	comm	0	0	⧖	0	0	1	2	0
	assoc	0	0	1	0	0	1	2	0
nat_plus_acc	comm	9	✗	⧖	⧖	⧖	0	2	0
	assoc	32	0	⧖	16	⧖	0	2	0
nat_times	comm	0	0	⧖	⧖	59	⧖	8	0
	assoc	0	0	1	⧖	⧖	⧖	12	0
nat_times_acc	comm	43	✗	⧖	⧖	⧖	2	2	2
	assoc	⧖	✗	⧖	⧖	⧖	✗	–	✗
nat_times_alt	comm	46	0	⧖	⧖	⧖	⧖	2	6
	assoc	⧖	✗	⧖	⧖	⧖	⧖	–	✗
nat_times_weird	comm	⧖	✗	⧖	⧖	⧖	⧖	–	✗
	assoc	⧖	✗	⧖	⧖	⧖	⧖	–	✗

Analysis. Our parameter analysis reveals that Propel's deductive power comes from two features. First, Propel' property generation approach (Section 2.4) is able to generate the auxiliary properties needed for most proofs for algebraic properties. The number of properties that Propel needs to generate for a successful proof ("Propel (aux #)" column) of some of the more complicated benchmarks (i.e., which other provers fail to verify), is a one-digit number for most cases, indicating that proving algebraic properties requires discovering auxiliary properties while, with Propel's property generation, only a small number of auxiliary properties needs to be checked.

Second, Propel can reason about both equalities and inequalities and the contradictions that can be derived from both. Most cases which Propel with a disabled inequality set ("Propel (no ≠)" column) fail to prove can also not be verified by the other provers. Thus, we believe that not only recording which expressions were identified to be equal, but also tracking which expressions were discovered *unequal* offers significant opportunities for improving the performance of theorem provers. Reasoning about algebraic properties is the key for populating the inequality set, as irreflexive, antisymmetric and connected relations dedicate that their arguments are unequal.

5 RELATED WORK

For an overview of theorem proving, we point the reader to the survey [31]. In this section, we focus on work dedicated specifically to algebraic properties, inductive theorem provers, *inductionless* induction theorem provers, and expressive type systems that enable proving program properties.

Correct Algebraic Properties. Servois [5] iteratively refines a starting hypothesis to create conditions under which two functions commute and two functions commute when the conditions are always true. Aleen and Clark [3] developed a static analysis technique that probabilistically determines if an imperative function commutes with itself based on how it modifies a memory layout. In contrast to verifying a function's commutativity, Gélineau [21] designed a Haskell library where special functions are commutative by design. by encapsulating into a monad that imposes

an order on the applied values, rendering their original order unobservable. Yet, it's unclear how to ensure other algebraic properties, such as associativity, using the methods above.

Inductive Theorem Provers. Zeno uses algebraic data types to establish the induction hypothesis [43, 44]. The tool checks Haskell code for properties specified in the code. In instances where a proof gets stuck, Zeno conjectures and attempts to prove additional lemmas. The lemma generation technique is derived from the *generalization* step used in Boyer-Moore inductive theorem provers [9] by changing recurrent subexpressions in an equation into an equation where the subexpression occurrences are replaced by a variable. Zeno then treats the altered equation as a speculative proposition to be verified independently. To avoid excessive generalization and eliminate wrong conjectures early, Zeno tries to find counterexamples before attempting any proof.

HipSpec [15, 17, 46] is another inductive theorem prover for Haskell. Instead of applying generalization to formulate lemmas during proof generation as Zeno does, HipSpec constructs a repository of proven lemmas that can potentially aid in the proving process before starting the proof process. It starts by *enumerating* all equalities between Haskell expressions up to a certain depth which undergo many filters. First, all equalities containing untypeable expressions are immediately discarded. Second, all those equating expressions of different types are also discarded. Third, it discards those that QuickCheck [14] can discover counterexamples for. Fourth, it uses the Z3 SMT solver [20] to prove the equalities that do not require induction. And finally, the expressions that remain are passed on to HipSpec's internal automatic inductive theorem prover for a final round of filtering. In practical applications, HipSpec managed to verify that widely-used implementations of specific type classes adhere to type class laws [4].

TheSy [42] employs equality graphs [32] (also known as program expression graphs) to effectively select a canonical representative from a class of equivalent programs for each expression. Similar to HipSpec, TheSy enumerates possible lemmas that may help the proof. TheSy enumerates lemmas by filtering out equivalent programs with the help of equality graphs, thus eliminating the need to re-prove them. MATHsAiD [29] is similar to HipSpec, aiding mathematicians in their exploration of theories. The tool employs program synthesis techniques to guide the generation of lemmas [52].

Lemma Enumeration and Generalization. The auxiliary properties proposed by Propel are both *enumerated* like HipSpec's and TheSy's lemmas, and are a product of *generalization* like Zeno and Boyer-Moore provers. Yet, as described in Section 2.4, enumerations follow a strict distributivity template that produces few expressions, and generalizations are a result of a few data constructor unfoldings of function arguments.

Like HipSpec and TheSy, Propel pre-compiles a list of potential auxiliary properties that may be helpful but limits enumeration to the local context of each function and relevant data constructors, while both HipSpec and TheSy extend their enumeration to all symbols in the environment. While Propel may overlook certain useful auxiliary properties involving two functions with a distant relationship, it tends to finalize proofs faster and – based on our evaluation – more effectively for the properties we are investigating. In addition, while HipSpec asserts lemmas prior to any proof attempt, Propel performs this process for each function. Hence auxiliary properties about upcoming functions are not conjectured until the proofs for the current function have been finalized.

Inductionless Induction. Inductive theorem provers, grouped under the terminology of "inductionless induction" or "proof by consistency", implicitly address induction [18, 50]. This approach has been revived in the field of cyclic proof theory [10, 45], and is used in automated theorem prover systems such as Cyclist [11]. An extension of GHC, CycleQ, incorporates a theorem prover to facilitate equational reasoning [23].

Another development is the integration of support for induction into the CVC4 SMT solver, which achieves this by skolemizing the inductive hypothesis [7, 37]. The SMT solver identifies a model that aligns with the negation of the inductive hypothesis and fails if the theorem is true.

However, all these systems handle properties as assertions. They do not manage them in types, like Propel does. The advantage of Propel's approach of tracking properties at the type level is that it enables higher-order functions to enforce properties about their inputs.

Type Systems. Dependent type systems such as Coq's calculus of constructions [19] or Agda's type system [8] have the ability to encode functional properties by lifting proof terms to the type level. Yet, this approach requires the user to manually construct such proof terms. In contrast, we focus on automating algebraic properties verification.

Our previous work on Propel [53] uses a type system to verify the implementations of CRDTs [41]. focusing only on the properties that are necessary in such domain. Instead, in this work, we apply Propel to a diverse range of domains and allow developers to express properties – such as type class laws – that are not built into the prover.

Refinement types, as in LiquidHaskell (LH), enable developers to attach propositions to data types [40, 47, 51]. While LH delegates the verification of the refinement predicates linked to types to an SMT solver, LiquidHaskell with Refinement Reflection (LH+RR) empowers programmers to prove propositions by creating proof objects that testify to the truth of these propositions [48]. This result can be achieved manually or by invoking the "Proof by Logical Evaluation (PLE)" proof-search algorithm. Though PLE can automate some proofs, it requires the developer to provide the structure of the induction by detailing the arguments to the recursive call.

Liu et al. [26] extended LiquidHaskell and Refinement Reflection to attach laws to type class definitions, which are expressed as members of the type class that instances must implement [47, 48]. Propel differs from Type Class Refinements (TCR) in two primary ways: First, Propel addresses predefined algebraic properties, resulting in a higher degree of automation. Second, to assert algebraic properties of functions in TCR, the functions must be represented as a data type and therefore defunctionalized [38]. This leads to higher-order functions becoming type functions parameterized by the defunctionalization identifiers, and necessitates manual handling of closures. As algebraic properties are directly in a function's type, no such transformation is needed in Propel.

With our solution, we aim to offer abstractions to developers that are not more heavyweight than needed for our use case. A dependent type system is very expressive but it often requires the programmers to manipulate proof terms manually. On the other hand, we have found LiquidHaskell not expressive enough, especially in its lack of quantification – which is consistent with its need of remaining decidable. We aim for the sweet spot where proofs are automatic while still supporting quantification (which commutativity, for example, requires).

6 CONCLUSION

Algebraic laws on functions are crucial in mathematics and often serve as a foundation for reasoning about computations. Despite playing such crucial role in abstract thinking, these laws are often overlooked in software programming practice. For instance, commutativity and associativity are key in the correctness of compiler optimizations, big data and data flow processing, distributed algorithms, data structures, but most languages lack built-in mechanisms to ensure and check the adherence of operations to these laws.

In this paper, we proposed Propel, a specialized verifier designed for algebraic laws that checks the adherence of functions to these laws in code. The key insight is Propel's ability to conjecture auxiliary properties and to reason about both equalities and inequalities of expressions, which is crucial to prove the law when other competitors do not succeed. We evaluated Propel across various domains and properties and demonstrated that our approach outperforms existing tools.

ACKNOWLEDGMENTS

The authors wish to thank the anonymous reviewers of PLDI 2024 for their valuable comments. This work is supported by the Swiss National Science Foundation (SNSF), grant 200429.

ARTIFACT AVAILABILITY

The artifact is available on Zenodo [54]. It includes the implementation of Propel as discussed in Section 3 with the induction rules. The implementation provides our Scala DSL that can be imported and used in Scala code, and a standalone verifier that checks the properties of functions implemented in a LISP dialect which is also described in the artifact. Moreover, all the benchmarks provided in Section 4 have dedicated scripts which can be executed to verify our reported results. The included README file provides a guide on how to interpret the output of the benchmarks.

REFERENCES

[1] [n. d.]. Apache Beam. https://beam.apache.org/. Accessed: 2013-07-12.

[2] Martín Abadi and Michael Isard. 2015. Timely dataflow: A model. In *Formal Techniques for Distributed Objects, Components, and Systems: 35th IFIP WG 6.1 International Conference, FORTE 2015, Held as Part of the 10th International Federated Conference on Distributed Computing Techniques, DisCoTec 2015, Grenoble, France, June 2-4, 2015, Proceedings 35*. Springer, 131–145.

[3] Farhana Aleen and Nathan Clark. 2009. Commutativity Analysis for Software Parallelization: Letting Program Transformations See the Big Picture. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems* (Washington, DC, USA) *(ASPLOS XIV '09)*. ACM, New York, NY, USA, 241–252. https://doi.org/10.1145/1508244.1508273

[4] Andreas Arvidsson, Moa Johansson, and Robin Touche. 2019. Proving Type Class Laws for Haskell. In *Trends in Functional Programming*, David Van Horn and John Hughes (Eds.). Springer International Publishing, Cham, 61–74. https://doi.org/10.1007/978-3-030-14805-8_4

[5] Kshitij Bansal, Eric Koskinen, and Omer Tripp. 2018. Automatic Generation of Precise and Useful Commutativity Conditions. In *Tools and Algorithms for the Construction and Analysis of Systems (Lecture Notes in Computer Science)*, Dirk Beyer and Marieke Huisman (Eds.). Springer International Publishing, Cham, 115–132. https://doi.org/10.1007/978-3-319-89960-2_7

[6] Haniel Barbosa, Clark 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* (Munich, Germany). Springer-Verlag, Berlin/Heidelberg, Germany, 415–442. https://doi.org/10.1007/978-3-030-99524-9_24

[7] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. 2011. CVC4. In *Computer Aided Verification*, Ganesh Gopalakrishnan and Shaz Qadeer (Eds.). Springer-Verlag, Berlin/Heidelberg, Germany, 171–177. https://doi.org/10.1007/978-3-642-22110-1_14

[8] Ana Bove, Peter Dybjer, and Ulf Norell. 2009. A Brief Overview of Agda – A Functional Language with Dependent Types. In *Theorem Proving in Higher Order Logics*, Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel (Eds.). Springer-Verlag, Berlin/Heidelberg, Germany, 73–78. https://doi.org/10.1007/978-3-642-03359-9_6

[9] R. S. Boyer, M. Kaufmann, and J. S. Moore. 1995. The Boyer-Moore Theorem Prover and Its Interactive Enhancement. *Computers & Mathematics with Applications* 29, 2 (Jan. 1995), 27–62. https://doi.org/10.1016/0898-1221(94)00215-7

[10] James Brotherston. 2005. Cyclic Proofs for First-Order Logic with Inductive Definitions. In *Automated Reasoning with Analytic Tableaux and Related Methods*, Bernhard Beckert (Ed.). Springer-Verlag, Berlin/Heidelberg, Germany, 78–92. https://doi.org/10.1007/11554554_8

[11] James Brotherston, Nikos Gorogiannis, and Rasmus L. Petersen. 2012. A Generic Cyclic Theorem Prover. In *Asian Symposium on Programming Languages and Systems*, Ranjit Jhala and Atsushi Igarashi (Eds.). Springer-Verlag, Berlin/Heidelberg, Germany, 350–367. https://doi.org/10.1007/978-3-642-35182-2_25

[12] P. Buneman, S. Davidson, and A. Kosky. 1992. Theoretical Aspects of Schema Merging. In *Proc. Int'l. Conf. on Extending Database Technology*. Vienna, Austria.

[13] Yu-Fang Chen, Lei Song, and Zhilin Wu. 2016. The Commutativity Problem of the MapReduce Framework: A Transducer-Based Approach. In *Computer Aided Verification*, Swarat Chaudhuri and Azadeh Farzan (Eds.). Springer International Publishing, Cham, 91–111. https://doi.org/10.1007/978-3-319-41540-6_6

[14] Koen Claessen and John Hughes. 2000. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00)*. SCM, New York, NY, USA, 268–279. https://doi.org/10.1145/351240.351266

[15] Koen Claessen, Moa Johansson, Dan Rosen, and Nick Smallbone. 2012. HipSpec: Automating Inductive Proofs of Program Properties. 16–5. https://doi.org/10.29007/3qwr

[16] 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. https://doi.org/10.1007/978-3-319-20615-8_23

[17] Koen Claessen, Nicholas Smallbone, and John Hughes. 2010. QuickSpec: Guessing Formal Specifications Using Testing. In *Tests and Proofs*, Gordon Fraser and Angelo Gargantini (Eds.). Springer-Verlag, Berlin/Heidelberg, Germany, 6–21. https://doi.org/10.1007/978-3-642-13977-2_3

[18] Hubert Comon. 2001. *Inductionless Induction.* North-Holland, Amsterdam, Chapter 14, 913–962. https://doi.org/10.1016/B978-044450813-3/50016-3

[19] Thierry Coquand and Gérard Huet. 1986. The Calculus of Constructions.

[20] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer-Verlag, Berlin/Heidelberg, Germany, 337–340. https://doi.org/10.1007/978-3-540-78800-3_24

[21] Samuel Gélineau. 2010. Commutative Composition: a conservative approach to aspect weaving. https://escholarship.mcgill.ca/concern/theses/gq67jr62t

[22] Márton Hajdú, Petra Hozzová, Laura Kovács, Johannes Schoisswohl, and Andrei Voronkov. 2020. Induction with Generalization in Superposition Reasoning. In *Intelligent Computer Mathematics*, Christoph Benzmüller and Bruce Miller (Eds.). Springer International Publishing, Cham, 123–137. https://doi.org/10.1007/978-3-030-53518-6_8

[23] Eddie Jones, C.-H. Luke Ong, and Steven Ramsay. 2022. CycleQ: An Efficient Basis for Cyclic Equational Reasoning. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (San Diego, CA, USA) *(PLDI '22)*. ACM, New York, NY, USA, 395–409. https://doi.org/10.1145/3519939.3523731

[24] Simon Peyton Jones. 2003. *Haskell 98 language and libraries: the revised report.* Cambridge University Press.

[25] Milind Kulkarni, Donald Nguyen, Dimitrios Prountzos, Xin Sui, and Keshav Pingali. 2011. Exploiting the commutativity lattice. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (32rd PLDI'11)*. ACM Press (NY), San Jose, CA, USA, 542–555.

[26] Yiyun Liu, James Parker, Patrick Redmond, Lindsey Kuper, Michael Hicks, and Niki Vazou. 2020. Verifying Replicated Data Types with Typeclass Refinements in Liquid Haskell. *Proceedings of the ACM on Programming Languages* 4, OOPSLA, Article 216 (Nov. 2020), 30 pages. https://doi.org/10.1145/3428284

[27] M. Snir, W. Otto, S. Huss-Lederman, D.W. Walker and J. Dongarra. 1996. MPI: The Complete Reference. MIT Press.

[28] Luca De Martini, Alessandro Margara, Gianpaolo Cugola, Marco Donadoni, and Edoardo Morassutto. 2023. The Noir Dataflow Platform: Efficient Data Processing without Complexity. arXiv:2306.04421 [cs.DC]

[29] Roy L McCasland, Alan Bundy, and Patrick F Smith. 2017. MATHsAiD: Automated mathematical theory exploration. *Applied Intelligence* 47, 3 (2017), 585–606. https://doi.org/10.1007/s10489-017-0954-8

[30] Derek G Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. 2013. Naiad: a timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. 439–455.

[31] M. Saqib Nawaz, Moin Malik, Yi Li, Meng Sun, and M. Ikram Ullah Lali. 2019. A Survey on Theorem Provers in Formal Methods. arXiv:1912.03028 [cs.SE]

[32] Charles Gregory Nelson. 1980. *Techniques for Program Verification.* Ph. D. Dissertation. Stanford, CA, USA. AAI8011683.

[33] The University of Glasgow. 2023. Prelude. https://hackage.haskell.org/package/base-4.19.0.0/docs/Prelude.html. Last accessed on 16 Nov 2023.

[34] W. Pottenger. 1998. The Role of Associativity and Commutativity in the Detection and Transformation of Loop-Level Parallelism. ACM press, New York, 188–195.

[35] Rachel Pottinger and Philip A. Bernstein. 2009. Associativity and Commutativity in Generic Merge. In *Conceptual Modeling: Foundations and Applications - Essays in Honor of John Mylopoulos (Lecture Notes in Computer Science, Vol. 5600)*, Alexander Borgida, Vinay K. Chaudhri, Paolo Giorgini, and Eric S. K. Yu (Eds.). Springer, 254–272.

[36] Rachel A. Pottinger and Philip A. Bernstein. 2003. - Merging Models Based on Given Correspondences. In *Proceedings 2003 VLDB Conference*, Johann-Christoph Freytag, Peter Lockemann, Serge Abiteboul, Michael Carey, Patricia Selinger, and Andreas Heuer (Eds.). Morgan Kaufmann, San Francisco, 862–873. https://doi.org/10.1016/B978-012722442-8/50081-1

[37] Andrew Reynolds and Viktor Kuncak. 2015. Induction for SMT Solvers. In *Verification, Model Checking, and Abstract Interpretation*, Deepak D'Souza, Akash Lal, and Kim Guldstrand Larsen (Eds.). Springer-Verlag, Berlin/Heidelberg, Germany, 80–98. https://doi.org/10.1007/978-3-662-46081-8_5

[38] John C. Reynolds. 1972. Definitional Interpreters for Higher-Order Programming Languages. In *Proceedings of the ACM Annual Conference – Volume 2* (Boston, Massachusetts, USA) *(ACM '72)*. ACM, New York, NY, USA, 717–740. https://doi.org/10.1145/800194.805852

[39] Martin C. Rinard and Pedro C. Diniz. 1997. Commutativity Analysis: A New Analysis Technique for Parallelizing Compilers. *ACM Trans. Program. Lang. Syst* 19, 6 (1997), 942–991.

[40] John Rushby, Sam Owre, and Natarajan Shankar. 1998. Subtypes for Specifications: Predicate Subtyping in PVS. *IEEE Transactions on Software Engineering* 24, 9 (Sept. 1998), 709–720. https://doi.org/10.1109/32.713327

[41] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011. Conflict-Free Replicated Data Types. In *Stabilization, Safety, and Security of Distributed Systems*, Xavier Défago, Franck Petit, and Vincent Villain (Eds.). Springer-Verlag, Berlin/Heidelberg, Germany, 386–400. https://doi.org/10.1007/978-3-642-24550-3_29

[42] Eytan Singher and Shachar Itzhaky. 2021. Theory Exploration Powered by Deductive Synthesis. In *Computer Aided Verification*, Alexandra Silva and K. Rustan M. Leino (Eds.). Springer International Publishing, Cham, 125–148. https://doi.org/10.1007/978-3-030-81688-9_6

[43] Willam Sonnex, Sophia Drossopoulou, and Susan Eisenbach. 2011. Zeno: A tool for the automatic verification of algebraic properties of functional programs.

[44] William Sonnex, Sophia Drossopoulou, and Susan Eisenbach. 2012. Zeno: An Automated Prover for Properties of Recursive Data Structures. In *Tools and Algorithms for the Construction and Analysis of Systems*, Cormac Flanagan and Barbara König (Eds.). Springer-Verlag, Berlin/Heidelberg, Germany, 407–421. https://doi.org/10.1007/978-3-642-28756-5_28

[45] Christoph Sprenger and Mads Dam. 2003. On the Structure of Inductive Reasoning: Circular and Tree-Shaped Proofs in the μCalculus. In *Foundations of Software Science and Computation Structures*, Andrew D. Gordon (Ed.). Springer-Verlag, Berlin/Heidelberg, Germany, 425–440. https://doi.org/10.1007/3-540-36576-1_27

[46] Irene Lobo Valbuena and Moa Johansson. 2015. Conditional Lemma Discovery and Recursion Induction in Hipster. *Electronic Communications of the EASST* 72 (2015).

[47] Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon Peyton-Jones. 2014. Refinement Types for Haskell. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming* (Gothenburg, Sweden) *(ICFP '14)*. ACM, New York, NY, USA, 269–282. https://doi.org/10.1145/2628136.2628161

[48] Niki Vazou, Anish Tondwalkar, Vikraman Choudhury, Ryan G. Scott, Ryan R. Newton, Philip Wadler, and Ranjit Jhala. 2017. Refinement Reflection: Complete Verification with SMT. *Proceedings of the ACM on Programming Languages* 2, POPL, Article 53 (Dec. 2017), 31 pages. https://doi.org/10.1145/3158141

[49] William E. Weihl. 1988. Commutativity-Based Concurrency Control for Abstract Data Types. *IEEE Trans. Computers* 37, 12 (1988), 1488–1505. http://doi.ieeecomputersociety.org/10.1109/12.9728

[50] Claus-Peter Wirth. 2005. *History and Future of Implicit and Inductionless Induction: Beware the Old Jade and the Zombie!* Springer-Verlag, Berlin/Heidelberg, Germany, 192–203. https://doi.org/10.1007/978-3-540-32254-2_12

[51] Hongwei Xi and Frank Pfenning. 1998. Eliminating Array Bound Checking through Dependent Types. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation* (Montreal, Quebec, Canada) *(PLDI '98)*. ACM, New York, NY, USA, 249–257. https://doi.org/10.1145/277650.277732

[52] Weikun Yang, Grigory Fedyukovich, and Aarti Gupta. 2019. Lemma Synthesis for Automating Induction over Algebraic Data Types. In *Principles and Practice of Constraint Programming*, Thomas Schiex and Simon de Givry (Eds.). Springer International Publishing, Cham, 600–617. https://doi.org/10.1007/978-3-030-30048-7_35

[53] George Zakhour, Pascal Weisenburger, and Guido Salvaneschi. 2023. Type-Checking CRDT Convergence. *Proc. ACM Program. Lang.* 7, PLDI, Article 162 (jun 2023), 24 pages. https://doi.org/10.1145/3591276

[54] George Zakhour, Pascal Weisenburger, and Guido Salvaneschi. 2024. *Automated Verification of Fundamental Algebraic Laws*. https://doi.org/10.5281/zenodo.10949342

Received 2023-11-16; accepted 2024-03-31