Type-Checking CRDT Convergence

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

Conflict-Free Replicated Data Types (CRDTs) are a recent approach for keeping replicated data consistent while guaranteeing the absence of conflicts among replicas. For correct operation, CRDTs rely on a merge function that is commutative, associative and idempotent. Ensuring that such algebraic properties are satisfied by implementations, however, is left to the programmer, resulting in a process that is complex and error-prone. While techniques based on testing, automatic verification of a model, and mechanized or handwritten proofs are available, we lack an approach that is able to verify such properties on concrete CRDT implementations.

In this paper, we present Propel, a programming language with a type system that captures the algebraic properties required by a correct CRDT implementation. The Propel type system deduces such properties by case analysis and induction: sum types guide the case analysis and algebraic properties in function types enable induction for free. Propel's key feature is its capacity to reason about algebraic properties (a) in terms of rewrite rules and (b) to derive the equality or inequality of expressions from the properties. We provide an implementation of Propel as a Scala embedding, we implement several CRDTs, verify them with Propel and compare the verification process with four state-of-the-art verification tools. Our evaluation shows that Propel is able to automatically deduce the properties that are relevant for common CRDT implementations found in open-source libraries even in cases in which competitors timeout.

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

Additional Key Words and Phrases: Conflict-Free Replicated Data Types, Type Systems, Verification

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

1 INTRODUCTION

Distributed systems replicate data to different machines for scalability and fault tolerance. Replication, however, raises the issue of keeping data consistent among the different replicas. On one side of the spectrum are approaches that provide strong consistency to ensure all replicas always hold the exact same copy of the replicated state [Ellis and Gibbs 1989]. Strong consistency incurs heavy coordination overhead and high latencies. Network partitions even make it infeasible to keep data consistent across different partitions without blocking data access completely. On the other side of the spectrum are systems that sacrifice strong consistency for availability and improved latency [Vogels 2009]. By allowing the data on different replicas to be changed independently, however, the state of the replicas might diverge over time and has to be reconciled to make changes *eventually* become visible everywhere. In particular, reconciling diverged states requires to resolve conflicts, i.e., when the same data was changed to different values on different replicas independently.

Authors' address: George Zakhour, george.zakhour@unisg.ch; Pascal Weisenburger, pascal.weisenburger@unisg.ch; Guido Salvaneschi, guido.salvaneschi@unisg.ch, University of St. Gallen, Torstrasse 25, St. Gallen, SG, 9000, Switzerland.

2475-1421/2023/6-ART162
https://doi.org/10.1145/3591276

Hence, to make sure that data is both available and (eventually) consistent, developers need to design proper conflict resolution schemes, which is error-prone and hard to implement [Kleppmann and Beresford 2017; Shapiro et al. 2011b]. Thus, recent research focuses on Conflict-Free Replicated Data Types (CRDTs), which avoid conflicts *by design* and provide strong eventual consistency, i.e., not only will all updates be observed eventually on all replicas, but two replicas that have seen the same set of updates will be in the same state [Shapiro et al. 2011b].

To avoid conflicts, state-based CRDTs use clever encodings to represent specific data structures, such that the state space of the encoding forms a semilattice. A large amount of recent work presents such encoding for various data structures [Almeida et al. 2018; Bieniusa et al. 2012; Kleppmann 2017; Kleppmann and Beresford 2017; Nicolaescu et al. 2015; Shapiro et al. 2011a]. A semilattice $\langle S, \sqcup \rangle$ is defined by a binary operation $\sqcup$ on its carrier set S that is commutative, associative and idempotent. In the context of CRDTs, this operation is often called *merge* and enables that the (diverged) state of two replicas can be joined automatically. Hence, two or more CRDTs in a different state can always converge to the merge of their states.

Ensuring convergence is even more critical since CRDTs cannot be easily composed. Hence, to take advantage of CRDTs, programmers have two choices: resorting to an existing CRDT for a data structure that fits their use case (if it exists), or implementing their own CRDT. Either way, the implementation must provide a correct merge function for the underlying semilattice.

Programmers have different ways to check these properties, like contracts, software testing or formally verifying a model by translating to formats readable by model checkers or SMT solvers. Research on new CRDTs typically resorts to manually proving convergence, or presents informal arguments based on pseudo code. Such approaches do not cover the complete input space, operate on a model instead of on the program directly, or require extensive manual work to provide proofs. Hence, ensuring the convergence of CRDTs is ultimately left to the programmer. Whereas advanced static type systems can provide a variety of correctness guarantees, we lack mechanisms that consider the algebraic properties required for CRDT convergence.

We propose a technique that proves the properties relevant to CRDTs automatically and *directly on the implementation*. We accomplish this by providing a type system that is able to reason about such algebraic properties, in particular about commutativity, associativity and idempotence. Thus, for a definition that is given the type of a commutative function, the type checker guarantees that it is commutative. Conversely, when encountering a function that has a commutative type, the type checker can use that fact to deduce algebraic properties of the code using the function. Thus, the type system approach enables deriving properties of the composition of functions.

To check an algebraic property of a function, our type system constructs a property derivation tree (a proof for the property) in addition to the usual typing derivation. To this end, our type system contains a proof engine, tailored to these algebraic properties. The exploration of possible trees is not complete due to the large search space. Still, our approach outperforms existing solutions that can prove algebraic properties for CRDT convergence directly on the implementation [Claessen et al. 2012; Sonnex et al. 2012] in terms of properties proven within a timeout. In contrast to Propel, the other approaches lack the ability to reason about algebraic properties and the (in)equality of expressions they induce. In summary, this paper makes the following contributions:

- We design Propel, a language with a type system tailored to ensure convergence of CRDTs.
- We implement Propel as an embedding into Scala (Section 3).
- We formally present the Propel type system that guarantees functions obey specified algebraic properties and prove its soundness, i.e., that derived properties do hold (Section 4).
- We evaluate Propel on common CRDT implementations found in open-source projects and show that it can to prove the relevant algebraic properties of these CRDTs automatically,

whereas existing approaches that allow expressing such properties and verifying them directly on the implementation lack algebraic reasoning and fail to derive them (Section 5).

2 MOTIVATING EXAMPLE

As a running example, we use a *Grow-only Counter (GCounter)*, a common CRDT, which is a replicated counter that can be incremented on different replicas independently from each other without the need for synchronization: The state of the GCounter replicas might be temporarily different until the state is merged. While a GCounter is a relatively simple CRDT, it showcases the compositionality issues that demand a type-based solution and how verifying CRDTs requires to reason about the interaction of the algebraic properties of the functions and relations involved.

A straightforward implementation of a GCounter's state is a list of numbers where the index in the list represents the replica and the value at a given index represents the current count for the respective replica. Hence, GCounters store a list of the current count for each replica. Merging two states zips the lists (to pairwise combine the counts that refer to the same replica) and takes the maximum for each pair (ensuring the every increment is counted exactly once):

```
def mergeGCounter(x: List[Num], y: List[Num]): List[Num] = zipWith(max)(x, y)
```

From the definition of `mergeGCounter`, it is not immediately clear whether such function implements a valid merge operation. In particular, a valid merge operation needs to be commutative (and also associative and idempotent). Whether `mergeGCounter` is commutative depends on whether the arguments `x` and `y` commute for `zipWith(max)` (defined later in Listing 2); and whether the arguments commute depends on which function is applied to `zipWith` as its first argument – here `max`. In fact, `x` and `y` commute for `zipWith` if and only if its first argument is a commutative function – which `max` is. Hence, to deduce commutativity under function composition, we need an approach that tracks commutativity properties of functions.

Clearly, we need to apply this kind of reasoning on algebraic properties of (compositions of) functions to all possible return values (i.e., for all branches a function can take). For illustration, consider the implementation of `max` (Listing 1) used in the `mergeGCounter` example – the implementation contains a subtle programming error that is not caught by standard type systems.

Listing 1. Faulty definition of `max`.

```
1  enum Num:
2    case Zero; case Bit0(num: Num); case Bit1(num: Num)
3
4  def max(x: Num, y: Num): Num = (x, y) match
5    case (Zero, y) => y
6    case (x, Zero) => x
7    case (Bit0(x), Bit0(y)) => Bit0(max(x, y))
8    case (Bit1(x), Bit1(y)) => Bit1(max(x, y))
9    case (Bit0(x), Bit1(y)) =>
10     if equals(max(x, y), y) then Bit0(y) else Bit0(x)
11   case (Bit1(x), Bit0(y)) =>
12     if equals(max(x, y), x) then Bit1(x) else Bit1(y)
```

Deriving commutativity of `max` requires case analysis on the different branches. In particular, which branch is taken depends on the `equals` function (Lines 10 and 12). Thereby, commutativity of `max` relies on the relational properties of `equals`, specifically reflexivity, symmetry and antisymmetry (i.e., an equality relation). Hence, tracking properties of functions and relations and analyzing the branches they lead to are at the core of our approach. The discussion on commutativity extends analogously to the other properties important for CRDT merge functions, i.e., associativity and idempotence.

Since checking whether functions satisfy certain properties is essential for ensuring CRDT convergence, we argue that these properties should receive the same level of correctness guarantees that a static type system provides. Ideally, a type system should catch the bug in the `max` implementation that makes it non-commutative. We will present a correct version in Listing 3.

3 PROPEL

In this section, we introduce the design of Propel with a type system that allows specifying algebraic properties in the type of functions. While the surface of Propel is intentionally minimal – the

only distinctive feature exposed to the developer is the ability to specify such function properties – Propel's core is the typing derivation proving that the properties hold. As usual, if a specified property cannot be deduced for a given function, the program is rejected by the type system.

Section 4 presents the type system in a core calculus. Instead, the examples in this section use an embedding of Propel into Scala, where algebraic properties of binary functions are expressed by a type of the form `P :=(A, A) =>: B`. `P` are the properties of the function and `A` and `B` the types of the function arguments and the result. For example, a Propel function on numbers that asserts commutativity of its arguments has type `Comm :=(Num, Num) =>: Num`. The introduction form for Propel functions in the embedding is the `prop[FunctionType]` construct (or `prop.rec` for recursive functions). Technically, `prop` is a macro that invokes the Propel type checker. For example, the following is a commutative, associative and idempotent version of the `mergeGCounter` function:

```
def mergeGCounter = prop[(Comm & Assoc & Idem) := (List[Num], List[Num]) =>: List[Num]] { zipWith(max) }
```

The `mergeGCounter` function type-checks since Propel can successfully derive the stated properties. We describe how algebraic properties of functions are derived in the following sections.

3.1 The Propel Framework for Property Deduction

This section introduces the core ideas of Propel's type checking. For functions, Propel first deduces the algebraic properties, e.g., $\texttt{f(x, y)} \stackrel{?}{=} \texttt{f(y, x)}$ for commutativity of a function `f`. Relations are represented by functions that return booleans. Once a property is proved, Propel uses the fact that it holds in all use sites of the function, e.g., `f(x, y)` can be rewritten to `f(y, x)` if `f` is commutative.

Table 1 shows the properties currently supported by Propel, which consist of (a) the essential algebraic properties discussed so far to prove CRDT convergence and (b) relational properties that are typically important to decide which branch a function takes. Relational properties enable the type system to reason about equality (which we will discuss later under *Equalities*).

The column *Equations to prove* shows the equations that need to be proven to ensure a property holds, e.g., $\texttt{f(x, y)} \stackrel{?}{=} \texttt{f(y, x)}$ for commutativity. To prove such an equation, Propel first expands the body of the function `f`. Then, Propel tries to derive a sequence of rewrites of the equation that leads to both sides being syntactically identical. Such rewriting process is a proof that the property holds for `f`. If the type system fails to find such a sequence, it rejects the program.

Propel is sound but incomplete, i.e., it may not be able to derive that a property holds even though it does but deriving a property guarantees that it holds. Since exploring all possible rewrites is infeasible, the exploration is restricted to a fixed number of "best" terms (according to an heuristics) after every rewrite. The concrete algorithm is in Section 3.4, whereas the formalization in Section 4 proves the soundness of our rewrites without imposing a specific algorithm to find a rewrite chain.

Propel constructs a property derivation tree (similar to typing derivation trees for standard type checking), which mixes applications of *rewrite rules* and *case analysis* and maintains a set of *equalities* that have been discovered to hold.

Rewrite rules. Our rewrite rules retain the equivalence of terms in the sense that terms evaluate to the same value for all arguments before and after the rewrite. Rewrites operate on open terms, where free variables symbolically represent arbitrary values of the correct type. The rules include

- the evaluation rules of the operational semantics extended to open terms,
- the algebraic rules that hold for a specific function as given in the *Derived equalities and inequalities* column, (e.g., commutativity for a function `f` allows rewriting `f(x, y)` to `f(y, x)` for arbitrary terms `x` and `y`) and
- the ability to rewrite a term `x` to a term `y` if `x` and `y` were discovered to be equal. Equality of terms can be discovered by case analysis or relational properties that induce equality.

Table 1. Propel's algebraic properties of binary functions $\circ$ and relations R.

Property	Equations to prove ($\stackrel{?}{=}$)	Derived equalities ($=$) and inequalities ($\neq$)
Commutativity	$x \circ y \stackrel{?}{=} y \circ x$	$\triangleright\ x \circ y = y \circ x$
Associativity	$(x \circ y) \circ z \stackrel{?}{=} x \circ (y \circ z)$	$\triangleright\ (x \circ y) \circ z = x \circ (y \circ z)$
Selection	$x \circ y \stackrel{?}{=} x$ or $x \circ y \stackrel{?}{=} y$	$\triangleright\ x \circ y = x$ or $x \circ y = y$
Idempotence	$x \circ x \stackrel{?}{=} x$	$\triangleright\ x \circ x = x$
Reflexivity	$xRx \stackrel{?}{=} \top$	$\triangleright\ xRx = \top$ $xRy = \bot \ \triangleright\ x \neq y$
Irreflexivity	$xRx \stackrel{?}{=} \bot$	$\triangleright\ xRx = \bot$ $xRy = \top \ \triangleright\ x \neq y$
Symmetry	$xRy \to yRx \stackrel{?}{=} \top$	$xRy = \top \ \triangleright\ yRx = \top$ $xRy = \bot \ \triangleright\ yRx = \bot$
Antisymmetry	$xRy \to \neg yRx \stackrel{?}{=} \top$ for $x \neq y$	$xRy = \top \ \triangleright\ yRx = \bot,\ x \neq y$ or $x = y$
Connectedness	$xRy \vee yRx \stackrel{?}{=} \top$ for $x \neq y$	$xRy = \bot \ \triangleright\ yRx = \top,\ x \neq y$ or $x = y$
Transitivity	$xRy \wedge yRz \to xRz \stackrel{?}{=} \top$	$xRy = \top,\ yRz = \top \ \triangleright\ xRz = \top$

Case analysis. We perform case analysis on pattern matches (with which also `if-then-else` can be expressed). Pattern matching with more than one case results in a branch in the property derivation tree, i.e., Propel proves that the property holds for every branch. When branching, Propel collects equalities and inequalities that hold in every branch. For example, when pattern-matching on a variable `x` with the patterns `Bit0(y)`, `Bit1(y)` and `Zero`, we add `x = Bit0(y)`, `x ≠ Bit1(y)` and `x ≠ Zero` to the set of (in)equalities in the first branch.

Equalities. Propel keeps track of expressions that were discovered equal. While there is no equality check built-in as a language construct, there are two ways to deduce that two expressions are equal. First, for pattern matching, unification of patterns with the scrutinee binds variables to expressions, hence variables and bound expressions are treated equal (as described before under *Case analysis*). Second, some relational properties (namely reflexivity, irreflexivity, antisymmetry and connectedness, cf. Table 1) dictate their elements are equal or unequal. For example, if a relation `r` is reflexive and we know `r(x, y) = false` (in some branch), we deduce `x ≠ y` (in the same branch).

Based on both options, we maintain a set of expressions known to be equal and a set of expressions known to be unequal. The role of the equality sets is twofold. First, equalities define rewrite rules, i.e., expressions that are equal can be rewritten into each other. Second, they enable *reductio ad absurdum* when both the equality and the inequality of the same two expressions is known. Similarly, deriving an equation which is provably false due to different data constructors is a contradiction, i.e., `Bit0(x) = Bit1(x)`; clearly, a branch where this equality would hold can never be taken.

3.2 Type-Based Deduction of Algebraic and Relational Properties

Table 1 provides an overview of the treatment of (in)equalities for each property supported by Propel. The *Equations to prove* column provides the equation that needs to be checked to derive that a specific property holds. An "or" means it is sufficient to derive either equation. The *Derived equalities and inequalities* provides the (in)equalities we can derive and the rewrite rules we can apply knowing that a property holds for a certain function. An "or" means both possibilities need to be examined, i.e., this situation constitutes a branch in the property derivation tree. $E \triangleright F$ means that, given E is a subset of the equality set, we can derive the equalities F.

The only way to verify the properties of recursive functions requires reasoning by induction. In our system, using an induction hypothesis at the recursive call amounts to using the rewrite rules induced by the function's properties. Because these are tracked in the type of the function they are available at the recursive call. Therefore no special treatment is needed for induction and we get it

for free under the assumption that the function terminates. Proof techniques that treat induction implicitly are known as *inductionless induction* or *proof by consistency* [Comon 2001; Wirth 2005]. Techniques to reject potentially nonterminating recursive functions exist [Barthe et al. 2006, 2004; Giménez 1995] which, we assume, are used on recursive functions when proving properties.

Algebraic properties. As an example, consider the definition of `zipWith` (Listing 2) used in `mergeGCounter`. The type parameter `T` denotes the type of the elements in the lists to zip; the type parameter `P >: (Comm & Assoc & Idem)` can be instantiated to any combination of the three properties. The first argument of `zipWith` is a function `f` of type `P := (T, T) =>: T`, i.e., a binary operation on values of type `T` with the algebraic properties `P`, the arguments `x` and `y` are of type `List[T]`. Further, `zipWith` has the same properties `P` as `f`, i.e., by the type `P := (List[T], List[T]) =>: List[T]`, `zipWith` is commutative, associative or idempotent if `f` is.

Listing 2. Definition of `zipWith`.

```
1 def zipWith[P >: (Comm & Assoc & Idem), T] =
2   prop.rec[(P := (T, T) =>: T) => (P := (List[T], List[T]) =>: List[T])]:
3     zipWith => f =>
4       case (Nil, y) => y
5       case (x, Nil) => x
6       case (x :: xs, y :: ys) => f(x, y) :: zipWith(f)(xs, ys)
```

Thus, we need to derive each property for `zipWith` under the assumption that `f` has the same property. For commutativity, we set up the equation `zipWith(f)(x, y)` $\stackrel{?}{=}$ `zipWith(f)(y, x)`, which needs to hold for all `f`, `x` and `y`. A case analysis on the branches of `zipWith` shows that commutativity trivially holds for `x = Nil` since the expressions reduce to `y` $\stackrel{?}{=}$ `y` and for `y = Nil` since the expressions reduce to `x` $\stackrel{?}{=}$ `x`. The only interesting case is `f(x, y) :: zipWith(f)(xs, ys)` $\stackrel{?}{=}$ `f(y, x) :: zipWith(f)(ys, xs)`. Since, the types of both `f` and `zipWith` assert commutativity, we can swap the order of the `x` and `y` and the `xs` and `ys` arguments in both calls. The last case illustrates how we can reason about recursive definitions without explicit induction by lifting algebraic properties to the type. Rewriting one side of the equation leads to an expression that is syntactically identical to the one on the other side. Similar derivations can be constructed for associativity and idempotence.

Relational properties. The conditions on which functions branch are often formulated in terms of relations, e.g., ordering or equality. We revisit the initial example of the (now corrected and property-type-checked version of the) `max` function on numbers (Listing 3). We observe that the last two branches check whether `max(x, y)` equals `y` or `x`, respectively. While commutativity is straightforward, we need additional knowledge of the `equals` relation to derive associativity. Being an equality relation, `equals` (Listing 4) is reflexive, symmetric and antisymmetric. From these properties follows that, in Line 9 (Listing 3), `max(x, y) = x` in the `then`-branch and `max(x, y)` $\neq$ `x` in the `else`-branch.

Listing 3. Definition of `max`.

```
1 def max = prop.rec[(Comm & Assoc & Idem) := (Num, Num) =>: Num]: max =>
2   case (Zero, y) => y
3   case (x, Zero) => x
4   case (Bit0(x), Bit0(y)) => Bit0(max(x, y))
5   case (Bit1(x), Bit1(y)) => Bit1(max(x, y))
6   case (Bit0(x), Bit1(y)) =>
7     if equals(max(x, y), y) then Bit1(y) else Bit0(x)
8   case (Bit1(x), Bit0(y)) =>
9     if equals(max(x, y), x) then Bit1(x) else Bit0(y)
```

Listing 4. Definition of `equals`.

```
1 def equals =
2   prop.rec[(Refl & Sym & Antisym) := (Num, Num) =>: Boolean]: equals =>
3     case (Zero, Zero) => true
4     case (Bit0(x), Bit0(y)) => equals(x, y)
5     case (Bit1(x), Bit1(y)) => equals(x, y)
6     case _ => false
```

To derive associativity of `max`, we postulate `max(Bit0(x), max(Bit1(y), Bit0(z)))` $\stackrel{?}{=}$ `max(max(Bit0(x), Bit1(y)), Bit0(z))`. The interesting case is when the left-hand side matches Line 7, and the right-hand side matches Line 9. On the left-hand side, the `if` condition on `equals(max(x, y), y)` requires case analysis: First, the case `max(x, y)` $\neq$ `y` lets us rewrite the equation `max(Bit0(x), Bit1(y)) = Bit1(y)` to `Bit0(x) = Bit1(y)`, which is a contradiction. Hence, we know this case cannot be

reached; analogously for `max(y, z)` ≠ `y` on the right-hand side. Second, the case `max(x, y)` = `y` on the left-hand side of the equation and `max(y, z)` = `y` on the right-hand both reduce to `Bit1(y)`, proving the equation holds. The remaining cases of `max` are proven using the same kind of reasoning.

3.3 Deduction of Auxiliary Properties

In practice, successful property deduction often depends on additional properties specific to a certain function not captured by the set of algebraic properties presented so far (in Table 1). Hence, Propel tries to discover such auxiliary properties. To this end, before attempting to deduce a specified algebraic property, Propel conjectures a set of potentially useful equations for a function and tries to prove them. Successfully proven equations are added to set of expressions that are known to be equal. The required auxiliary properties can subtly depend on the way a function is defined.

Listing 5. Variants of Peano number addition.

```
1  enum Nat:
2    case Zero; case Succ(pred: Nat)
3
4  def add2p =
5    prop.rec[(Comm & Assoc) := (Nat, Nat) =>: Nat]: add2p =>
6      case (Zero, y) => y
7      case (Succ(x), y) => Succ(add2p(x, y))
8
9  def add3p =
10   prop.rec[(Comm & Assoc) := (Nat, Nat) =>: Nat]: add3p =>
11     case (Zero, y) => y
12     case (x, Zero) => x
13     case (Succ(x), Succ(y)) => Succ(Succ(add3p(x, y)))
```

For example, for addition of Peano numbers (Listing 5), deducing commutativity for `add2p` does not rely on additional properties. Deducing commutativity for `add3p`, however, requires a statement about how the successor constructor `Succ` can be moved outwards from the function's arguments, i.e., `add3p(Succ(x), y)` = `Succ(add3p(x, y))` and `add3p(x, Succ(y))` = `Succ(add3p(x, y))`.

Proving a conjecture can fail because it is false or because its proof requires another conjecture not proven yet. Hence, once a conjecture is proven, it is added to the set of known equalities and can be used to prove other conjectures from then on. Thus, when a new conjecture is proven, Propel retries proofs of failed conjectures until all conjectures are proven or no progress is made.

Note that conjectures, however they look like, can never impair the soundness of the prover, since any conjecture requires a proof before being used. Therefore, the specific conjecture generation approach is not part of Propel's core calculus. It is, however, fundamental for being able to prove the desired algebraic properties. Hence, we describe Propel's techniques to discover equations next.

Case Analysis and Generalization. To derive auxiliary properties, we perform a case analysis on the function we are examining, based on the different possible input values. To this end, for every argument which is of an algebraic data type, we generate possible values. For Peano numbers, for instance, possible values are `Zero`, `Succ(Zero)`, `Succ(Succ(Zero))`, etc. We found that is often enough to unfold every recursive type only once, i.e., generating `x`, `Zero` and `Succ(x)` for fresh variables `x`. The case analysis for the different argument values (a) yields the expression of the branch that the function takes for the given arguments and (b) refines the arguments further based on the patterns matched to take the respective branch. The result of the case analysis is a set of equations where the left-hand side is the function applied to the arguments and the right-hand side is the expression of the branch taken for the given arguments. While the equations derived this way are trivially provable, we apply the following two generalizations to obtain further conjectures.

First, if the right-hand side is the same as one of the arguments on the left-hand side, we replace both by a variable. For example, `add(Zero, Succ(y))` $\stackrel{?}{=}$ `Succ(y)` is generalized to `add(Zero, y)` $\stackrel{?}{=}$ `y`.

Second – as the left-hand side has the form $f\ a_0 \ldots a_n$ by construction – for all variables that appear free in both the left- and right-hand side, we check (i) if they appear only within a single argument a_i of f on the left-hand side, and (ii) if a_i has the same type as the result of f. If this is the case, we generate (a) left-hand sides where all arguments except a_i are generalized to variables and (b) right-hand sides where an arbitrary subexpression e_j that matches the type of f is replaced

Algorithm 1 Property Deduction

```
1:  procedure Deduce(f)
2:      C ← AuxiliaryEquations(f)                ▷ Section 3.3
3:      C ← C ∪ AlgebraicEquations(f)            ▷ Table 1
4:      P ← ∅
5:
6:      repeat
7:          P' ← ∅
8:          for all p ∈ C do
9:              if Verify(p, P) then
10:                 P' ← P' ∪ {p}
11:         C ← C \ P'
12:         P ← P ∪ P'
13:     until P' = ∅ ∨ C = ∅
14:
15:     if AscribedProperties(f) ⊈ P then
16:         return PropertyDeductionError
17:     else
18:         return P
```

Algorithm 2 Property Verification

```
1:  procedure Verify(p, P)
2:      return ⋀_{(p',E')∈Cases(p,P)} Equal(p', P, E')
3:
4:  procedure Equal(p, P, E)
5:      R ← Rewrite(p, P, E)
6:      { p' } ← Top(R, 1)
7:      (e0 ≟? e1) ← p
8:      (e'0 ≟? e'1) ← p'
9:      return Contradiction(E)
10:             ∨ e'0 = e'1
11:             ∨ (p ≠ p' ∧ Verify(p', P, E))
12:
13: procedure Rewrite(t, P, E)
14:     t† ← ToSubterms(t)
15:     t‡ ← { t' ∈ Rewrite(t†, P, E) }
16:     t‡† ← { FromSubterms(t') | t' ∈ t‡ }
17:     t‡‡ ← ⋃_{t'∈t‡†} ApplyEquations(t', P, E)
18:     return Top(t‡‡, N)
```

by the function call f of the left-hand side with a_i replaced by e_j, i.e., $f\ a_0 \ldots e_j \ldots a_n$. For example, a generalization of `add(Zero, Succ(y))` $\stackrel{?}{=}$ `Succ(y)` is `add(x, Succ(y))` $\stackrel{?}{=}$ `Succ(add(x, y))`.

While the equations derived this way in principle constitute possible rewrites in two directions, we restrict them to rewrites from the left-hand to the right-hand side since that direction is always a simplification of the term for generalizations constructed as described above – either eliminating a function call or moving data constructors outwards.

Distributivity. A notable algebraic property that we ignored until now is distributivity. The distinctive feature of distributivity in contrast to the properties discussed so far is that it involves multiple functions. To account for distributive relationships among different functions, we collect all other functions `g` that are used in the body of the examined function `f` and that are closed over the same type and conjecture distributive properties, e.g., `g(x, f(y, z))` $\stackrel{?}{=}$ `f(g(x,y), g(x,z))` and `f(x, g(y, z))` $\stackrel{?}{=}$ `g(f(x,y), f(x,z))`.

3.4 Property Deduction, Algorithmically

Our type system internally distinguishes between equalitites $\mathcal{E}$ of the form $e_0 = e_1$ (or inequalities $e_0 \neq e_1$) and properties $\mathcal{P}$, which are quantified equalities, e.g., $\forall x, y.\ \texttt{add}\ x\ y = \texttt{add}\ y\ x$. Additional equalitites can arise during case analysis while properties do not.

To deduce the properties of a function f (Algorithm 1), we first construct a set of conjectured properties C that contains generalized cases, distributivity properties (Line 2) and the algebraic equations which fit the type of f (Line 3). Thus, we also examine algebraic properties that are not ascribed as part of the type of f since they might be useful to prove other properties, e.g., commutativity is often needed to prove associativity. Proven properties are removed from the set of conjectures C (Line 11) and added to the set of properties $\mathcal{P}$ (Line 12), which is used to prove other properties (Line 9). We attempt proofs iteratively until no more properties can be proven (Lines 6 to 13). Finally, we check that the derived properties $\mathcal{P}$ conform to the explicitly ascribed properties (Line 15). A successful deduction may yield a superset of the ascribed properties, i.e., the algorithm may infer additional properties of the functions and add them to the function's type. Note that the core calculus formalizes type-checking, not inference.

To check whether a property p of form $e_0 \stackrel{?}{=} e_1$ holds (Algorithm 2), we perform a case analysis (Line 2) on both sides of the equality of p and on the expressions in $\mathcal{E}$. We then check whether e_0

equals e_1 for all cases. We leave out the definition of Cases for brevity. The function returns a set of $(p, \mathcal{E})$ pairs where p is the equation to prove for every case and $\mathcal{E}$ the set of equalities that hold in the respective case. Equalities are derived from variable bindings in pattern matches and algebraic laws (as defined in Table 1). To check equality, we rewrite p based on the equalities $\mathcal{E}$ and the properties $\mathcal{P}$ (Line 5). Since various sequences of rewrite rule applications exists, Rewrite returns a set of equations. We compare the "top" equation according to an arbitration order on terms (Line 6). The equality check succeeds if there is a contradiction in the equality set, i.e., the respective branch can never be taken (Line 9), both sides of the equality are syntactically equal (Line 10), or applying the rewrite rules produced a new $e_0' \stackrel{?}{=} e_1'$ and e_0' equals e_1' (Line 11). Rewriting a term recursively applies the rewrite rules to all elements in the list of subterms (Line 15). This yields a list of sets of rewritten terms for every subterm, which we sequentialize into a set of lists to reassemble the rewritten subterms (Line 16). The ApplyEquations function (definition is left out) recursively applies all matching rewrite rules in $\mathcal{P}$ and $\mathcal{E}$ to the reassembled term (Line 17). Since a complete exploration of the space of all terms equal under the rewrite rules explodes combinatorially, we only retain a fixed number of terms for each subterm (Line 18).

Naturally, it is desirable to discover a sequence of rewrites that proves the given equation. As a trade-off between performance and exploring relevant rewrites, we traverse the complete syntax tree of the equation bottom-up but only retain a set of bounded size for rewrite results for every subtree. The heuristic for which trees to keep uses a metric that favors "simpler" trees, e.g., it favors data constructors over variables over applications over abstractions the closer they are to the tree's root. Further, ApplyEquations bounds to what extend rewrites can grow trees in the number of nodes. While our heuristics is relatively simple, it proved effective for the use case of checking algebraic properties of CRDTs. Propel's core, however, is not the exploration strategy but the ability to reason about algebraic properties and the (in)equalities and inequalities derived from them.

4 FORMALIZATION

This section presents Propel's core calculus: the syntax (Section 4.1); the reduction semantics (Section 4.2); the type system (Section 4.3) – including the proof rules for property annotations on functions (Section 4.4); and the main soundness theorem (Section 4.5) stating that :hen an expression is given a function type annotated with a property then the property holds. Finally, we show how our type system lifts (in)equalities defined by the user to the meta level (Section 4.6).

4.1 Syntax

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

Definition 1 (Syntax).

Constructors K	Data types X	Types $\mathsf{T} ::= \mathsf{X} \mid \mathsf{T}_1 \xrightarrow{\overline{\mathsf{p}}} \mathsf{T}_2$	Values $\mathsf{v} ::= \lambda_{\overline{\mathsf{p}}} \mathsf{x}{:}\mathsf{T}.\mathsf{e} \mid \mathsf{K}\ \overline{\mathsf{v}}$
Expressions	$\mathsf{e} ::= \mathsf{x} \mid \mathsf{e}_1\ \mathsf{e}_2 \mid \lambda_{\overline{\mathsf{p}}} \mathsf{x}{:}\mathsf{T}.\mathsf{e} \mid \mathsf{K}\ \overline{\mathsf{e}} \mid \mathsf{case}\ \mathsf{e}\ \{\overline{\mathsf{K}_\mathsf{i}\ \overline{\mathsf{x}_\mathsf{i}} \Rightarrow \mathsf{e}_\mathsf{i}}\} \mid \mathsf{fix}\ \mathsf{e}$		
Simple expressions	$\mathsf{t} ::= \mathsf{x} \mid \mathsf{K}\ \overline{\mathsf{t}}$		
Properties	$\mathsf{p} ::= \mathsf{comm} \mid \mathsf{assoc} \mid \mathsf{idem} \mid \mathsf{sel} \mid \mathsf{refl} \mid \mathsf{irefl} \mid \mathsf{sym} \mid \mathsf{antisym} \mid \mathsf{trans} \mid \mathsf{conn}$		

An expression is either a variable, an application, an abstraction annotated with a set of properties, a construction using a constructor K, a destruction expression that associates to some constructors and fresh variables an expression, and a fixed point operator that allows for recursion. Simple expressions only consist of constructors and variables. A type is either a data type X or a function annotated with some properties. (Recursive) data types are provided through the set X with their constructors in K. We do not represent recursive types as fixed points. This approach aligns with most real-world languages where data types are defined via dedicated constructs, e.g. `class`, `struct`,

or `enum` etc. Algorithmically, before type-checking, we collect all data types and constructors into X and K. The function properties that the syntax supports are: commutativity (comm), associativity (assoc), selectivity (sel) and idempotence (idem). Moreover, we represent each relation $R \subseteq \mathsf{T}_1 \times \mathsf{T}_2$ through its characteristic function $\mathsf{T}_1 \to \mathsf{T}_2 \to \mathbf{2}$ where $\mathbf{2}$ is the boolean type with its constructors $\top$ and $\bot$. We track these relational properties: reflexivity (refl), irreflexivity (irefl), symmetry (symmetry), antisymmetry (antisym), transitivity (trans), and connectedness (conn).

The notation $\overline{s}$ is for a list of zero or more items of a syntactic class s. Sometimes we introduce an index i under the line $\overline{s_i}$. When it is not clear from context where this index is bound, we use the notation $\overline{s}^i$. We use $\overline{s}^{i,p(i)}$ to restrict the list to the indexes satisfying $p(i)$.

4.2 Reduction

The operational semantics adopts an evaluation context C and rules of shape $\mathsf{e}_1 \to \mathsf{e}_2$. We denote the capture-avoiding substitution of a variable x for expression e_2 in expression e_1 with $\mathsf{e}_1[\mathsf{e}_2/x]$.

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

Definition 3 (Evaluation Rules).

$$\text{(E-App)}\ (\lambda_{\overline{\mathsf{p}}}\mathsf{x}{:}\mathsf{T}.\mathsf{e}_1)\ \mathsf{v} \to \mathsf{e}_1[\mathsf{v}/x] \qquad \text{(E-Case)}\ \mathsf{case}\ \mathsf{K_j}\ \overline{\mathsf{v_j}}\ \{\overline{\mathsf{K_i}\ \overline{\mathsf{x_i}} \Rightarrow \mathsf{e_i}}\} \to \mathsf{e_j}[\overline{\mathsf{v_j}/\mathsf{x_j}}]$$

$$\text{(E-Fix)}\ \mathsf{fix}\ \lambda_{\overline{\mathsf{p}}}\mathsf{x}{:}\mathsf{T}.\mathsf{e} \to \mathsf{e}[\mathsf{fix}\ \lambda_{\overline{\mathsf{p}}}\mathsf{x}{:}\mathsf{T}.\mathsf{e}/\mathsf{x}] \qquad \text{(E-Context)}\ \frac{\mathsf{e} \to \mathsf{e}'}{C[\mathsf{e}] \to C[\mathsf{e}']}$$

The evaluation context and rules define the standard call-by-value reduction semantics.

4.3 Type Checking

We first define the context used in our typing rules.

Definition 4 (Typing Contexts).

Variable Context	$\Gamma ::= \overline{\mathsf{x} : \mathsf{T}}$	Rule Context	$\mathcal{R} ::= \overline{\mathsf{r}}$
Data Type Context	$\mathcal{K} ::= \overline{\mathcal{K}_\mathsf{X}}$	Rule	$\mathsf{r} ::= \forall \overline{\mathsf{x} : \mathsf{T}}.\ \mathsf{e}_1 = \mathsf{e}_2$
Constructor Context	$\mathcal{K}_\mathsf{X} ::= \overline{\mathsf{K} : \overline{\mathsf{T}} \to \mathsf{X}}$	Typing Context	$\mathbb{\Gamma} ::= \mathcal{R}; \mathcal{K}; \Gamma$

The data type context $\mathcal{K}$ maps each data type X to a non-empty constructor context $\mathcal{K}_\mathsf{X}$ which maps constructors to a type $\overline{\mathsf{T}} \to \mathsf{X}$. When the list of arguments $\overline{\mathsf{T}}$ to a constructor is empty then it is mapped to the type X. To eliminate ambiguities among the data type constructed by each constructor, we assume that, for any two data types $\mathsf{X}_1 \neq \mathsf{X}_2$, constructors are distinct: $\mathrm{dom}\,\mathcal{K}_{\mathsf{X}_1} \cap \mathrm{dom}\,\mathcal{K}_{\mathsf{X}_2} = \emptyset$.

As function properties are handled at the type level, we require an additional context to track them. We further track any auxiliary properties (i.e., lemmas) that we derive as well. Syntactically these lemmas are quantified equalities, which we call rules. Rules are stored in the context $\mathcal{R}$.

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

We present the typing rules next.

Definition 5 (Typing Rules).

$$\text{(T-Var)}\ \frac{\mathsf{x} : \mathsf{T} \in \mathbb{\Gamma}}{\mathbb{\Gamma} \vdash \mathsf{x} : \mathsf{T}} \qquad \text{(T-Cons)}\ \frac{\mathsf{K} : \overline{\mathsf{T}} \to \mathsf{X} \in \mathcal{K}_\mathsf{X} \qquad \overline{\mathbb{\Gamma} \vdash \mathsf{e} : \mathsf{T}}}{\mathbb{\Gamma} \vdash \mathsf{K}\ \overline{\mathsf{e}} : \mathsf{X}} \qquad \text{(T-App)}\ \frac{\mathbb{\Gamma} \vdash \mathsf{e}_1 : \mathsf{T}_1 \xrightarrow{\overline{\mathsf{p}}} \mathsf{T}_2 \qquad \mathbb{\Gamma} \vdash \mathsf{e}_2 : \mathsf{T}_1' \qquad \mathsf{T}_1' \leq \mathsf{T}_1}{\mathbb{\Gamma} \vdash \mathsf{e}_1\ \mathsf{e}_2 : \mathsf{T}_2}$$

$$\text{(T-Abs)}\ \frac{\begin{array}{c}\overline{\mathsf{r} = \mathsf{p}_{\mathsf{T}_1}[\lambda_{\overline{p}}\mathsf{x}{:}\mathsf{T}_1.\mathsf{e}]} \\ \forall i.\llbracket\Gamma, \mathsf{r}_1 \ldots \mathsf{r}_{i-1}; \cdot; \cdot \Vdash \mathsf{r}_i \\ \llbracket\Gamma, \overline{\mathsf{r}}, \mathsf{x} : \mathsf{T}_1 \vdash \mathsf{e} : \mathsf{T}_2\end{array}}{\llbracket\Gamma \vdash \lambda_{\overline{p}}\mathsf{x}{:}\mathsf{T}_1.\mathsf{e} : \mathsf{T}_2}$$

$$\text{(T-Fix)}\ \frac{\llbracket\Gamma \vdash \mathsf{e} : (\mathsf{T}_1 \xrightarrow{\overline{p}} \mathsf{T}_2) \to \mathsf{T}_1 \xrightarrow{\overline{p}} \mathsf{T}_2}{\llbracket\Gamma \vdash \mathsf{fix}\ \mathsf{e} : \mathsf{T} \xrightarrow{\overline{p}} \mathsf{T}}$$

$$\text{(T-Lemma)}\ \frac{\mathsf{r} \in \mathit{lemma}(\lambda_{\overline{p}}\mathsf{x}{:}\mathsf{T}_1.\mathsf{e}) \quad \llbracket\Gamma; \cdot; \cdot \Vdash \mathsf{r} \qquad \llbracket\Gamma, \mathsf{r} \vdash \mathsf{fix}\ \lambda_{\overline{p}}\mathsf{x}{:}\mathsf{T}_1.\mathsf{e} : \mathsf{T}_2}{\llbracket\Gamma \vdash \mathsf{fix}\ \lambda_{\overline{p}}\mathsf{x}{:}\mathsf{T}_1.\mathsf{e} : \mathsf{T}_2}$$

$$\text{(T-Case)}\ \frac{\mathcal{K}_\mathsf{X} = \overline{\mathsf{K}_\mathsf{i} : \overline{\mathsf{T}_{1,\mathsf{i}}} \to \mathsf{X}} \qquad \dfrac{\llbracket\Gamma \vdash \mathsf{e} : \mathsf{X}}{\llbracket\Gamma, \overline{\mathsf{x}_\mathsf{i} : \mathsf{T}_{1,\mathsf{i}}} \vdash \mathsf{e}_\mathsf{i} : \mathsf{T}_2}}{\llbracket\Gamma \vdash \mathsf{case}\ \mathsf{e}\ \{\overline{\mathsf{K}_\mathsf{i}\ \overline{\mathsf{x}_\mathsf{i}} \Rightarrow \mathsf{e}_\mathsf{i}}\} : \mathsf{T}_2}$$

T-Var, T-Cons, T-App, and T-Fix are standard. T-Case ensures the type of the scrutinee expression is a data type. It checks that all the constructors in the data type's constructor context have a handler, and that every handler has the same type under the same typing context extended with each handler's bound variables. T-App uses the subtyping relation in Definition 6 that allows passing a function with more properties than expected to one that expects less.

T-Abs (1) translates each property into a rule as defined in Definition 7, (2) proves each translated rule using the $\Phi \Vdash \mathsf{r}$ relation (Section 4.4) such that each rule is allowed to use previously proved rules, and (3) proceeds to type the body of the function in the usual manner. To enable inductive reasoning, we assume the property holds when checking the function's body: during the proof of said property, the type system can use it at recursive calls when arguments get smaller. Using 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 as mentioned in Section 3.2.

Finally, T-Lemma uses the helper function *lemma* that we leave undefined. As long as *lemma* is decidable the soundness of Propel remains. We assume it computes a set of well-typed rules representing conjectures that must be proven before type-checking continues. Propel's implementation employs the lemma generation strategy of Section 3.3. The calculus, however, does not impose a specific strategy. Thus T-Lemma picks a rule, proves it, and type-checks the fixed point with the additional rule added to the context. Formally, T-Lemma represents the cut rule in formal logic.

Definition 6 (Subtyping Relation).

$$\text{(ST-Data type)}\ \frac{}{\mathsf{X} \leq \mathsf{X}} \qquad \text{(ST-Function)}\ \frac{\overline{\mathsf{p}_1} \supseteq \overline{\mathsf{p}_2} \qquad \mathsf{T}_{11} \geq \mathsf{T}_{21} \qquad \mathsf{T}_{12} \leq \mathsf{T}_{22}}{\mathsf{T}_{11} \xrightarrow{\overline{p_1}} \mathsf{T}_{12} \leq \mathsf{T}_{21} \xrightarrow{\overline{p_2}} \mathsf{T}_{22}}$$

Definition 7 (Translation From Properties to Rules).

$$\begin{array}{ll}
\mathsf{comm}_\mathsf{T}[\mathsf{e}] = \forall \mathsf{x}{:}\mathsf{T}, \mathsf{y}{:}\mathsf{T}.\ \mathsf{e}\ \mathsf{x}\ \mathsf{y} = \mathsf{e}\ \mathsf{y}\ \mathsf{x} & \mathsf{sym}_\mathsf{T}[\mathsf{e}] = \forall \mathsf{x}{:}\mathsf{T}, \mathsf{y}{:}\mathsf{T}.\ \mathsf{e}\ \mathsf{x}\ \mathsf{y} = \mathsf{e}\ \mathsf{y}\ \mathsf{x} \\
\mathsf{assoc}_\mathsf{T}[\mathsf{e}] = \forall \mathsf{x}{:}\mathsf{T}, \mathsf{y}{:}\mathsf{T}, \mathsf{z}{:}\mathsf{T}.\ \mathsf{e}\ \mathsf{x}\ (\mathsf{e}\ \mathsf{y}\ \mathsf{z}) = \mathsf{e}\ (\mathsf{e}\ \mathsf{x}\ \mathsf{y})\ \mathsf{z} & \mathsf{refl}_\mathsf{T}[\mathsf{e}] = \forall \mathsf{x}{:}\mathsf{T}.\ \mathsf{e}\ \mathsf{x}\ \mathsf{x} = \top \\
\mathsf{idem}_\mathsf{T}[\mathsf{e}] = \forall \mathsf{x}{:}\mathsf{T}.\ \mathsf{e}\ \mathsf{x}\ \mathsf{x} = \mathsf{x} & \mathsf{irefl}_\mathsf{T}[\mathsf{e}] = \forall \mathsf{x}{:}\mathsf{T}.\ \mathsf{e}\ \mathsf{x}\ \mathsf{x} = \bot
\end{array}$$

$$\mathsf{sel}_\mathsf{T}[\mathsf{e}] = \forall \mathsf{x}{:}\mathsf{T}, \mathsf{y}{:}\mathsf{T}, \mathsf{eq}{:}\mathsf{T} \xrightarrow{\text{refl,sym,antisym}} \mathsf{T} \to \mathsf{T}.\ \mathsf{case}\ \mathsf{eq}\ (\mathsf{e}\ \mathsf{x}\ \mathsf{y})\ \mathsf{x}\ \{\top \Rightarrow \top, \bot \Rightarrow \mathsf{eq}\ (\mathsf{e}\ \mathsf{x}\ \mathsf{y})\ \mathsf{y}\} = \top$$

$$\mathsf{antisym}_\mathsf{T}[\mathsf{e}] = \forall \mathsf{x}{:}\mathsf{T}, \mathsf{y}{:}\mathsf{T}, \mathsf{z}{:}\mathsf{T}.\ \mathsf{case}\ \mathsf{e}\ \mathsf{x}\ \mathsf{y}\ \{\bot \Rightarrow \mathsf{z}, \top \Rightarrow \mathsf{x}\} = \mathsf{case}\ \mathsf{e}\ \mathsf{y}\ \mathsf{x}\ \{\bot \Rightarrow \mathsf{z}, \top \Rightarrow \mathsf{y}\}$$

$$\mathsf{trans}_\mathsf{T}[\mathsf{e}] = \forall \mathsf{x}{:}\mathsf{T}, \mathsf{y}{:}\mathsf{T}, \mathsf{z}{:}\mathsf{T}.\ \mathsf{case}\ \mathsf{e}\ \mathsf{x}\ \mathsf{y}\ \{\bot \Rightarrow \top, \top \Rightarrow \mathsf{case}\ \mathsf{e}\ \mathsf{y}\ \mathsf{z}\ \{\bot \Rightarrow \top, \top \Rightarrow \mathsf{e}\ \mathsf{x}\ \mathsf{z}\}\} = \top$$

$$\begin{array}{rl}
\mathsf{conn}_\mathsf{T}[\mathsf{e}] = \forall \mathsf{x}{:}\mathsf{T}, \mathsf{y}{:}\mathsf{T}, \mathsf{z}{:}\mathsf{T}.\ & \mathsf{case}\ \mathsf{e}\ \mathsf{x}\ \mathsf{y}\ \{\top \Rightarrow \mathsf{z}, \bot \Rightarrow \mathsf{case}\ \mathsf{e}\ \mathsf{y}\ \mathsf{x}\ \{\top \Rightarrow \mathsf{z}, \bot \Rightarrow \mathsf{x}\}\} \\
& = \mathsf{case}\ \mathsf{e}\ \mathsf{x}\ \mathsf{y}\ \{\top \Rightarrow \mathsf{z}, \bot \Rightarrow \mathsf{case}\ \mathsf{e}\ \mathsf{y}\ \mathsf{x}\ \{\top \Rightarrow \mathsf{z}, \bot \Rightarrow \mathsf{y}\}\}
\end{array}$$

4.4 Property Checking

This section presents the proof rules used by T-Abs and T-Lemma.

Definition 8 (Proof Contexts).

Known Equalities	$\mathcal{E}^+ ::= \overline{\mathsf{e}_1 = \mathsf{e}_2}$	Proof Context	$\Phi ::= \mathcal{R}; \mathcal{K}; \Gamma\ \mathcal{E}^+; \mathcal{E}^-$
Known Inequalities	$\mathcal{E}^- ::= \overline{\mathsf{e}_1 \neq \mathsf{e}_2}$	Equality Context	$\mathbb{E} ::= [\,] = \mathsf{e}_2 \mid \mathsf{e}_1 = [\,]$

The proof system tracks equalities in $\mathcal{E}^+$ and inequalities in $\mathcal{E}^-$ which are used to reduce the goal to syntactic equality or to derive a contradiction. Overall, the proof rules have the form $\mathcal{R}; \mathcal{K}; \Gamma; \mathcal{E}^+; \mathcal{E}^- \Vdash r$, shortened to $\Phi \Vdash r$. To append rules, variable bindings, equalities, or inequalities we do it directly on Φ which is syntactically unambiguous. To avoid duplicating rules for left-hand-sides and right-hand-sides of equalities we introduce the equality context $\mathbb{E}$.

Proof rules. The proof rules codify the relationship between the proof context and the goal. They do not attempt to manipulate particular rules or (in)equalities.

Definition 9 (Proof Rules).

$$\text{(P-Refl)}\ \frac{}{\Phi \Vdash e = e} \qquad \text{(P-Derive)}\ \frac{\Phi \triangleright \Phi' \qquad \Phi' \Vdash r}{\Phi \Vdash r} \qquad \text{(P-Contra}^-\text{)}\ \frac{e \neq e \in \Phi}{\Phi \Vdash r}$$

$$\text{(P-Contra}^\pm\text{)}\ \frac{\begin{array}{c}e_1 = e_2 \in \Phi \\ e_1 \neq e_2 \in \Phi\end{array}}{\Phi \Vdash r} \qquad \text{(P-Contra}^+\text{)}\ \frac{\begin{array}{c}K_1\ \overline{e_1} = K_2\ \overline{e_2} \\ K_1 \neq K_2\end{array}}{\Phi \Vdash r} \qquad \text{(P-Contra}^{\circlearrowleft}\text{)}\ \frac{\begin{array}{c}x = K\ \bar{t} \\ \exists i.x \in t_i\end{array}}{\Phi \Vdash r}$$

$$\text{(P-Intro)}\ \frac{\Phi, \overline{x:T} \Vdash e_1 = e_2}{\Phi \Vdash \forall \overline{x:T}.\ e_1 = e_2} \qquad \text{(P-Abs)}\ \frac{\Gamma \vdash e : T \qquad \Phi, x:T, x = e \Vdash \mathbb{E}[x]}{\Phi \Vdash \mathbb{E}[e]}$$

$$\text{(P-Eq)}\ \frac{x = e \in \Phi}{\Phi \Vdash r} \qquad \text{(P-Case)}\ \frac{\begin{array}{c}\overline{\Phi, \overline{\overline{x_{ij} : T_{ij}}}, e = K_i\ \overline{x_i}, \overline{e \neq K_j\ \overline{x_j}}^{\,j, j \neq i}}^{\,i} \Vdash r \\ \Gamma \vdash e : X \qquad \mathcal{K}_X = \overline{K_i : \overline{T_i} \to X}\end{array}}{\Phi \Vdash r}$$

P-Refl states that two syntactically equal expressions are provably equal. P-Derive states that a rule is provable if it's provable under newly derived (in)equalities (Definition 10).

P-Contra$^-$ and P-Contra$^\pm$ conclude the proof using the principle of contradiction. Constructors construct non-overlapping values, thus P-Contra$^+$ is another form for contradiction. Finally P-Contra$^{\circlearrowleft}$ encodes that circular reasoning on simple terms is yet another contradiction.

P-Intro proves a quantification by introducing for each quantified variable a fresh one and proving the equality. P-Abs lifts one side of the goal into the equality context by assigning it to a fresh variable and proving the goal using that variable. P-Eq replaces in the goal and contexts every occurrence of a variable with an expression known to equate it.

P-Case implements case-analysis. If an expression is a data type, then for each constructor it proves the goal assuming that the expression is equal to said constructor and unequal to all others.

Derivation rules. The derivation rules encode the manipulation of known rules and (in)equalities.

Definition 10 (Derivation Rules).

$$\text{(D-Eq)}\ \frac{x = e \in \Phi}{\Phi \triangleright \Phi[e/x]} \qquad \text{(D-Abs)}\ \frac{\Gamma \vdash e : T}{\Phi \triangleright \Phi[x/e], x:T, x = e} \qquad \text{(D-Fun)}\ \frac{\lambda_{\overline{p}} x:T.e_1 = e_2 \in \Phi}{\Phi \triangleright \Phi, z:T, e_1[z/x] = e_2\ z}$$

$$\text{(D-Cons}^-\text{)}\ \frac{\begin{array}{c}K_1 \neq K_2 \\ e_1 = K_1 \overline{e_{11}} \in \Phi \\ e_2 = K_2 \overline{e_{22}} \in \Phi\end{array}}{\Phi \triangleright \Phi, e_1 \neq e_2} \qquad \text{(D-Arg}^-\text{)}\ \frac{e\ \overline{e_1}\ e_2\ \overline{e_3} \neq e\ \overline{e_1}\ e_2'\ \overline{e_3} \in \Phi}{\Phi \triangleright \Phi, e_2 \neq e_2'} \qquad \text{(D-NewR)}\ \frac{x:T_1 \xrightarrow{\overline{p}} T_2 \in \Phi}{\Phi \triangleright \Phi, \overline{p_{T_1}[x]}}$$

$$\text{(D-Cons}^+\text{)}\ \frac{K\overline{e_1} = K\overline{e_2} \in \Phi}{\Phi \triangleright \Phi, \overline{e_1 = e_2}} \qquad \text{(D-Rule)}\ \frac{\begin{array}{c}\forall \overline{x:T}.\ e_1 = e_2 \in \Phi \\ \overline{\Gamma \vdash x\sigma : T} \qquad \overline{x'} = \overline{x} \setminus \mathrm{dom}(\sigma)\end{array}}{\Phi \triangleright \Phi, \overline{x:T}, (e_1 = e_2)\sigma}$$

$$\text{(D-App)}\ \frac{\mathbb{E}[(\lambda_{\overline{p}} x:T.e_1)\ e_2] \in \Phi}{\Phi \triangleright \Phi, \mathbb{E}[e_1[e_2/x]]} \qquad \text{(D-Case)}\ \frac{\mathbb{E}[\mathrm{case}\ K_i\ \overline{e_i'}\ \{\overline{K_j\ \overline{x_j} \Rightarrow e_j}\}] \in \Phi}{\Phi \triangleright \Phi, \mathbb{E}[e_i[\overline{e_i'/x_i}]]}$$

D-Eq replaces all occurrences of a variable with an expression equal to it anywhere in the context. D-Abs replaces all occurrences of an expression by a fresh variable and records that the fresh variable is equal to that expression. D-Fun is functional extensionality: a function is equal to another if it is equal at every input. D-Cons$^-$ states that two expressions are unequal if they are constructions using different constructors. D-Arg$^-$ states that if a function application produces different results for all but one equal inputs, then the arguments at that one position are unequal. This rule is valid because every evaluation rule in the calculus is deterministic and functions are pure. D-NewR states that the properties of a function can be used. D-Cons$^+$ expresses that the arguments of equal constructions are equal if they use the same constructor. This rule is valid because constructors are injective. D-App derives performs β-reductions anywhere in the context. D-Case reduces a case expression whose scrutinee is a known constructor to the applicable cases.

D-Rule makes use of the rules derived from function annotations and conjectured by the *lemma* helper function. It states that if σ is a substitution of variables that substitutes some quantified variables for expressions of the same type, then we can derive a new equality, which is the rule's equality specialized to σ and the remaining quantified variables not in σ's domain introduced fresh.

4.5 Soundness

Our calculus enjoys type safety in terms of progress and preservation:

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

Theorem 2 (Preservation). *If* $\cdot;\mathcal{K};\cdot \vdash \mathsf{e} : \mathsf{T}$ *and* $\mathsf{e} \longrightarrow \mathsf{e}'$ *then* $\cdot;\mathcal{K};\cdot \vdash \mathsf{e}' : \mathsf{T}$.

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 then $\mathsf{e}\ \mathsf{x}\ \mathsf{y} = \mathsf{e}\ \mathsf{y}\ \mathsf{x}$ for every x, y.

To that end, we first define equality. The first option that presents itself is syntactic equality which we symbolize with $=$. Sadly it is too weak. For example, it does not allow us to express idempotence of the logical or function, a simple equality that we wish to prove: $\forall \mathsf{x}{:}\mathsf{2}.\mathsf{case}\ \mathsf{x}\ \{\top \Rightarrow \top, \bot \Rightarrow \mathsf{x}\} = \mathsf{x}$. Clearly, both terms are not syntactically equal. Yet, they do normalize to the same expression. Hence, a more suitable definition of equality is syntactic equality up to normal forms. Definition 11 presents a definition that conveys our intuition for equality at a type T, which we symbolize with $\equiv_{\mathsf{T}}$. It states that two expressions computing a data type are equal under the following conditions. First, they must both normalize to values which are syntactically constructions. Second, the constructors at the normal form must be the same. And third, the arguments passed to the constructors must be pair-wise equal under our definition of equality. If the two expressions compute a function, then they are equal if they normalize and produce equal outputs given arbitrary equal inputs.

Definition 11. Given a data type context $\mathcal{K}$ and two expressions $\mathsf{e}_1, \mathsf{e}_2$

- $\mathsf{e}_1 \equiv_{\mathsf{X}} \mathsf{e}_2$ if and only if $\cdot;\mathcal{K};\cdot \vdash \mathsf{e}_1 : \mathsf{X}$, $\cdot;\mathcal{K};\cdot \vdash \mathsf{e}_2 : \mathsf{X}$, $\mathsf{e}_1 \rightarrow^* \mathsf{K}\overline{\mathsf{v_1}}$, $\mathsf{e}_2 \rightarrow^* \mathsf{K}\overline{\mathsf{v_2}}$, $\mathcal{K}_{\mathsf{X}}(\mathsf{K}) = \overline{\mathsf{T}} \rightarrow \mathsf{X}$, and $\overline{\mathsf{v_1} \equiv_{\mathsf{T}} \mathsf{v_2}}$, and
- let $\mathsf{T} = \mathsf{T}_1 \xrightarrow{\overline{\mathsf{p}}} \mathsf{T}_2$ then $\mathsf{e}_1 \equiv_{\mathsf{T}} \mathsf{e}_2$ if and only if $\cdot;\mathcal{K};\cdot \vdash \mathsf{e}_1 : \mathsf{T}_1 \xrightarrow{\overline{\mathsf{p}}} \mathsf{T}_2$, $\cdot;\mathcal{K};\cdot \vdash \mathsf{e}_2 : \mathsf{T}_1 \xrightarrow{\overline{\mathsf{p}}} \mathsf{T}_2$, and assuming $\mathsf{e}_1 \rightarrow^* \lambda_{\overline{\mathsf{p}}}\mathsf{x} : \mathsf{T}_1.\mathsf{e}_{11}$ and $\mathsf{e}_2 \rightarrow^* \lambda_{\overline{\mathsf{p}}}\mathsf{x} : \mathsf{T}_1.\mathsf{e}_{22}$ then for any $\mathsf{e}_3, \mathsf{e}_4, \mathsf{T}_1{}'$ such that $\mathsf{e}_3 \equiv_{\mathsf{T}_1{}'} \mathsf{e}_4$ and $\mathsf{T}_1{}' \leq \mathsf{T}_1$ then $\mathsf{e}_{11}[\mathsf{e}_3/x] \equiv_{\mathsf{T}_2} \mathsf{e}_{22}[\mathsf{e}_4/x]$.

We may leave out the type annotation from $\equiv_{\mathsf{T}}$ when it is clear from the context.

To show that Definition 11 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* $\mathsf{e}_1 \longrightarrow \mathsf{e}_1{}'$ *then* $\mathsf{e}_1 \equiv_{\mathsf{T}} \mathsf{e}_2$ *if and only if* $\mathsf{e}_1{}' \equiv_{\mathsf{T}} \mathsf{e}_2$.

Lemma 2. $\equiv_{\mathsf{T}}$ *is an equivalence relation at well-typed expressions.*

Definition 12 defines 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 13 defines equivalent substitutions under $\mathbb{\Gamma}$ to be the ones that map every variable to equal expressions. Next, in Lemma 3, we prove that expressions equivalent under the same substitution are equivalent under equivalent substitutions.

Definition 12. Let $\mathbb{\Gamma}$ and σ be given. $\sigma \vDash \mathbb{\Gamma}$ when $\cdot; \mathcal{K}; \cdot \vdash \sigma(x) : T$ for every $x : T \in \mathbb{\Gamma}$.

Definition 13. 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* $e_1\sigma_1 \equiv e_2\sigma_2$ *then* $e_1\sigma_1 \equiv e_2\sigma_2$.

Armed with this library of lemmas we define a valid proof context to be one where the equalities and inequalities assuming the equalities in the rules hold where every equality is reinterpreted under the definition of equality in Definition 11.

Definition 14. A proof context Φ is 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 $r = \forall\overline{x : T}.e_1 = e_2$ then $r_{\equiv} = \overline{\lambda x : T}.e_1 = \overline{\lambda x : T}.e_2$,
- if $\mathcal{R} = \overline{r}$ then $\mathcal{R}_{\equiv} = \overline{r_{\equiv}}$, where the overline denotes conjunction.

We prove the following soundness theorems about the validity of the proof context. We prove that (a) a proof context derived from a valid one is itself valid, (b) every rule that is proved in a valid context holds, and (c) if a function can be typed then its properties hold. 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* $\Phi \Vdash r$, *then* $(r\sigma)_{\equiv}$.

Theorem 5. *If* $\mathbb{\Gamma} \vdash \lambda_{\overline{p}} x : T.e$ *and* $\sigma \vDash \mathbb{\Gamma}$ *and* $\mathcal{R}_{\equiv}\sigma$, *then* $\overline{p_{T_1}[\lambda_{\overline{p}} x : T.e\sigma]_{\equiv}}$.

Theorem 6. *If* $\cdot; \mathcal{K}; \cdot \vdash e : T_1 \xrightarrow{\overline{p}} T_2$ *and* e *normalizes, then* $\overline{p_{T_1}[e]_{\equiv}}$.

4.6 Equality Lifting

In this section we show that equality can be embedded in the language. Given the definition of a reflexive, symmetric, and antisymmetric relation, the proof system is able to lift that definition to the meta level as an equality, since equality is the unique relation with these properties, where the system is able to reason about equalities. This result justifies the choice of quantifying over a reflexive, symmetric, and antisymmetric function in the selection property in Definition 7.

Proposition 1. *Given a proof context* Φ *with* $\mathsf{eq} : T \xrightarrow{\text{refl,sym,antisym}} T \to T \in \Phi$ *and* $\mathsf{eq}\ a\ b = \bot \in \Phi$ *there exists a sequence of derivation* $\Phi \triangleright^n \Phi'$ *such that* $a \neq b \in \Phi'$.

Proposition 2. *Given a proof context* Φ *with* $\mathsf{eq} : T \xrightarrow{\text{refl,sym,antisym}} T \to T \in \Phi$ *and* $\mathsf{eq}\ a\ b = \top \in \Phi$ *there exists a sequence of derivation* $\Phi \triangleright^n \Phi'$ *such that* $a = b \in \Phi'$.

5 EVALUATION

To show the applicability of Propel to verifying CRDT convergence, we implemented a library of 20 CRDTs [Shapiro et al. 2011a] based on well-known designs from open source projects [Baquero 2014; Heinrichs 2017; Meiklejohn 2016; Rusu 2016; Sypytkowski 2018], from which we took the design specifications for our implementations. Our library also include variants of the same CRDT using either lists or maps of Peano numbers or bit vectors. We checked a total of 91 properties on the CRDTs and helper functions. The different variants are based on the following CRDTs:

- The *GCounter* (grow-only counter) is a counter that can only be incremented, not decremented. Internally, it is an array of numbers. The number at an index i in the array of replica j represents replica j's knowledge of i's current count.
- The *PNCounter* (positive-negative counter) is a counter that can be incremented and decremented. Internally, it is an array of pair of *GCounters*. The first *GCounter* of every pair counts the number of increments and the second *GCounter* counts the number of decrements.
- The *BCounter* (bounded counter) allows each replica only a fixed number of increments. Internally, it is an array of pairs with a *GCounter* and a matrix whose rows and columns represent the number of increments received from and sent to each replica, respectively.
- The *LWWReg* (last-write-wins register) contains a single value and a timestamp. The value is only updated when the update's timestamp is younger than the register's current one.
- A *GSet* (grow-only set) represents a set that only supports additions but not removals. Internally, it is a function from the type of elements to booleans, i.e., the set's characteristic function.
- An *ORSet* (observed-remove set) allows adding and removing values in any order. Internally, it is a pair of maps from the element of the set to a version vector. The domains in the pair are a set of added elements and a set of removed elements. A version vector is a *GCounter* that counts, per replica, the number of additions or removals of that element. Version vectors can be sorted lexicographically and provide an alternative to timestamps.
- A *2PSet* (two-phase set) allows adding and removing values. But, once removed, values cannot be added again. Like *PNCounter*'s relationship to *GCounter*, internally a *2PSet* is a pair of *GSets*.

Using our approach to type-check the Propel versions revealed that the original open-source implementations for *LWWReg* are not fully commutative. Precisely, when the two timestamps in an LWWReg are equal, a commutative, associative, and idempotent tie-breaking function must be applied to the value. The original specification of a LWWReg [Johnson and Thomas 1975] uses the maximum data value when the timestamps are equal. We found that the implementations do not handle that edge case and assume the timestamps are always distinct. While this assumption is likely to hold in the vast majority of cases, it can potentially violate consistency.

5.1 Verification of Algebraic Properties

To evaluate Propel's ability to prove properties of common CRDTs, we compare it against four state-of-the-art competitors. We focus on systems that are similar to Propel in the sense that they (a) allow expressing the algebraic properties of interest and (b) verify them directly on an implementation (or at least on a source-code-like representation).

We compare against four alternatives. Two are automated inductive theorem provers: HipSpec [Claessen et al. 2012] uses *theory exploration* for its lemma generation (a bottom-up approach that builds a library of lemmas before attempting any proof); Zeno [Sonnex et al. 2012] uses *lemma discovery by generalization* (synthesizing and proving lemmas on demand). Both parse Haskell code and attempt to prove the properties defined in it. We further compare against two SMT solvers capable of reasoning by induction: cvc5 [Barbosa et al. 2022] and Vampire [Hajdú et al. 2020]. To make the results between automated theorem provers and SMT solvers comparable, we reimplemented the used data types inductively – hence the SMT solvers cannot take advantage of some of their theories, e.g., for numbers, bit vectors, and sets, but still use some theories, e.g., for data types. This approach is in line with common benchmarks for theorem provers, such as TIP [Claessen et al. 2015], to set up a meaningful comparison between different provers. As the provers require a specific import format (like SMTLIB2 or Haskell), we could not run them directly on the available open-source implementations. We strove for implementations that are as similar as possible in Haskell (for HipSpec and Zeno), 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. A timeout is needed for two reasons. First, HipSpec's strategy of theory exploration amounts to producing every expression up to some depth and prove it before attempting the main property's proof, which leads to HipSpec taking up to hours to prove some properties. For example, the proofs for LWWReg took up to five hours. Second, SMT solvers may diverge while finding a counterexample. We aim for a verification that takes less than a minute. All provers are in native binaries, hence the measurements do not include VM startup.

Table 2. Verification of CRDT properties.

CRDT		Properties Proven By Prover *Order: commutativity, associativity, idempotence* *Time in seconds, timeout ⧗ or fail ✗*													
		HipSpec			Zeno			cvc5			Vampire			Propel		
GCounter	*(Peano numbers)*	2	2	2	0	0	0	⧗	⧗	11	⧗	⧗	0	1	1	1
GCounter	*(bit vectors)*	⧗	⧗	⧗	⧗	⧗	0	⧗	⧗	⧗	⧗	⧗	4	5	5	5
BCounter	*(Peano numbers)*	3	9	3	0	0	0	⧗	⧗	⧗	⧗	⧗	⧗	2	2	2
BCounter	*(bit vectors)*	⧗	⧗	⧗	⧗	⧗	0	⧗	✗	⧗	⧗	⧗	⧗	6	6	6
PNCounter	*(Peano numbers)*	2	2	2	0	0	0	⧗	⧗	⧗	⧗	⧗	9	1	1	1
PNCounter	*(bit vectors)*	⧗	⧗	⧗	⧗	⧗	0	⧗	⧗	⧗	⧗	⧗	⧗	5	5	5
LWW	*(Peano numbers)*	46	1	13	✗	✗	0	⧗	⧗	⧗	⧗	⧗	8	1	1	1
LWW	*(bit vectors)*	⧗	⧗	⧗	✗	⧗	0	⧗	⧗	✗	⧗	⧗	⧗	4	✗	4
GSet		2	2	2	✗	✗	✗							0	0	0
ORSet		⧗	⧗	⧗	✗	✗	✗							1	1	1
2PSet		2	2	2	✗	✗	✗							0	0	0

CRDTs. Table 2 shows the results for different provers. A ✗ indicates that the prover finished before the process was terminated but could not prove the property. A ⧗ indicates that the prover timed out. Otherwise, we report the verification time in seconds. The SMT solvers we used are not able to reason about higher-order functions. Therefore, we were unable to verify GSet, ORSet, and 2PSet, where sets are represented by functions. Hence, we leave their entries blank in the table.

CRDT helper functions and TIP benchmarks. In addition to the detailed results in Table 2, our evaluations includes further variants of CRDTs and helper functions. We plot the results for all properties in Figure 1. Successfully proven properties are green, proofs that timed-out are yellow, and failed proofs are red. Further, we applied Propel to the subset of the TIP 2015 benchmarks for inductive theorem provers [Claessen et al. 2015] that check algebraic properties (Figure 2).

Evaluation results. Our results show that existing approaches to directly verify the code fall short on proving the algebraic properties essential for CRDT convergence. Propel was able to automatically derive the desired properties for all CRDTs but one. The reason for that case is that our exploration of possible property derivation trees failed to find the tree that proves the property. This specific derivation tree could be discovered by a more sophisticated exploration algorithm. Yet, we leave tuning the algorithm to future work. Propel's core contribution that sets it apart from the other approaches lies in its ability to reason about algebraic properties and both the equalities and inequalities that follow from them. The TIP benchmarks mainly cover operations (such as addition, multiplication, min/max) and ordering relations on natural numbers, bit vectors and integers. Propel is able to prove one property (commutativity for addition on integers) more than the closest competitor HipSpec and far more than the other provers. The results indicate that our approach substantially improves over the other approaches and is suitable as a type system for algebraic correctness needed for CRDTs.

Case example: Variants of a simple function. To give an intuition of how sensitive the different approaches are with respect to a concrete implementation, we compare the simple addition of two Peano numbers in two variants (presented before in Listing 5): destruction over the first argument and destruction over both arguments. The results are in Table 3. The `add2p` variant

Table 3. Variants of Peano number addition.

Function	Properties Proven By Prover *Order: commutativity, associativity* *Time in seconds, timeout ⧗ or fail ✗*									
	HipSpec		Zeno		cvc5		Vampire		Propel	
add2p	2	2	1	1	0	0	0	0	0	0
add3p	44	28	1	✗	0	⧗	⧗	⧗	0	0

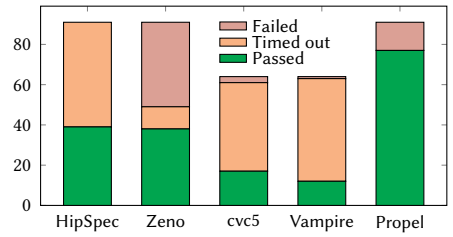

Fig. 1. Properties of CRDTs and helper functions.

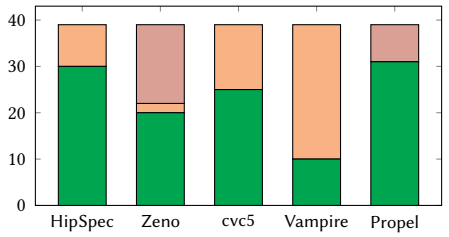

Fig. 2. Properties from TIP benchmarks.

is easier for the provers than `add3p` because the latter requires to derive the additional property that the successor constructor `Succ` can be moved outwards from one of the arguments of `add3p`.

6 DISCUSSION

Limitations. We cannot prove the properties of every merge function. As our evaluation shows, we were unable to prove associativity of the LWW register's merge function on bit vectors. This is because our heuristics failed to apply a rewrite rule that would have completed the proof. Generally, the heuristics tends to prune the set of expressions Propel explores early, speeding up the proofs but leaving potentially critical rewrites unexplored. We found this trade-off works well in practice. Yet, allowing the exploration of more rewrite chains or improving the (relatively simple) heuristics (Section 3.4) could make this proof discoverable.

Listing 6. A strange maximum.

```
1 def max =
2   prop.rec[(Comm & Assoc & Idem) := (Nat, Nat) =>: Nat]: max =>
3     case (Zero, Zero) => pred(max(Succ(x), Succ(y)))
4     case (x, Zero) => x
5     case (x, Succ(Zero)) => x
6     case (Zero, y) => y
7     case (Succ(Zero), y) => y
8     case (Succ(x), Succ(y)) => Succ(max(x,y))
```

Further, we found that very complex CRDTs like *Automerge* [Kleppmann 2017] (a JSON CRDT) are challenging to prove. None of the tools in our evaluation is able to prove them automatically. A foundational part of these complex CRDTs is often a variant of a *Replicated Growable Array (RGA)* [Shapiro et al. 2011a]. In its simplest form, an RGA is a list of elements, each tagged with its time of addition. Propel is unable to prove commutativity of the RGA merge as it does not generate the crucial conjecture `rgaMerge(xs, v :: ys)` $\stackrel{?}{=}$ `rgaMerge(v :: xs, ys)`, stating we can move the head from one list to the other. Similar lemmas would be required for merge sort, for instance.

It is also possible to construct a merge function whose typing judgment does not have a type derivation in our formal model. Consider a variation of `max` over the Peano numbers (Listing 6) where `pred` is the predecessor function. The rules of the type system cannot deduce that the branch on Line 3 computes `Zero` since it is impossible to apply the induction hypothesis rewrite at the recursive call because the arguments are not decreasing.

Counter-Example Discovery. When encountering a pattern match, Propel performs case analysis and proves each branch under the assumptions that the pattern matches the scrutinee expression. Further, it keeps track of these assumptions in the equality set. Hence, when a proof fails because the theorem is false, we can leverage the record of cases not only to report the case in which the proof failed but also to describe the class of inputs that makes the property false based on the patterns that led to the case.

Listing 7. TIP's bit vector multiplication.

```
1 enum Num:
2   case One; case ZeroAnd(num: Num); case OneAnd(num: Num)
3
4 def plus = prop.rec[Comm := (Num,Num) =>: Num]: /* ... */
5
6 def times = prop.rec[Comm := (Num,Num) =>: Num]: times =>
7   case (One, y) => y
8   case (ZeroAnd(x), y) => ZeroAnd(times(x, y))
9   case (OneAnd(x), y) => plus(ZeroAnd(times(x, y)),y)
```

For example, for the multiplication of bit vectors from the TIP benchmarks (Listing 7), Propel can disprove idempotence. Attempting to prove `times(x, x)` $\stackrel{?}{=}$ `x`, Propel performs case analysis. The first case to fail is when `x` is a `ZeroAnd`. Rewriting `ZeroAnd(times(x, ZeroAnd(x)))` $\stackrel{?}{=}$ `ZeroAnd(x)` using `times`'s commutativity, reducing the `times` application and applying the induction hypothesis yields `ZeroAnd(ZeroAnd(x))` $\stackrel{?}{=}$ `ZeroAnd(x)`. Eliminating common constructors produces the contradiction. Thus, the pattern `ZeroAnd(x)` describes a class of counterexamples which we can specialize to the smallest counterexample `ZeroAnd(One)`. Crucially, the equality set for the examined branch does not contain any equalities that may potentially constitute a contradiction.

Hence, in some situations, our approach is able to discover counterexamples as a byproduct of a proof attempt. Yet, we do not implement a dedicated search for counterexamples.

7 RELATED WORK

Correctness of CRDTs. Zeller et al. [2014] define a framework to verify CRDTs using Isabelle/HOL [Nipkow et al. 2002] based on a model defined by the data type and the merge, update, and query functions. The framework by Nagar and Jagannathan [2019] verifies the correctness of a model using Z3 under different consistency schemes which allows some CRDTs to be proven consistent under specified network conditions when they might not be under weaker conditions. Similarly, Boogie [Nair et al. 2020] uses an SMT solver to verify a CRDT specification resembling an implementation with manually-added invariants and assertions throughout. The previous approaches do not verify implementations, like we do, but only models.

Instead of verifying the CRDT (model), other researchers have investigated CRDTs correctness *by construction.* Weidner et al. [2020] tackle the composability CRDTs. They apply the semi-direct product of groups in abstract algebra on CRDTs so that the semi-direct product of two CRDTs becomes a correct CRDT by construction. Katara [Laddad et al. 2022] is a tool to synthesize a correct CRDT given a sequential implementation of the datatype. By specifying orderings on its operations, the developer is able to control conflict resolution. The synthesis of a CRDT from a datatype with conflicts must choose a conflict resolution scheme. This choice cannot be customized without jeopardizing correctness guarantees. Moreover they are limited to a special language or require the user to provide orderings on the operations.

Inductive Theorem Provers. An approach to use algebraic data types for constructing the induction hypothesis was proposed in Zeno [Sonnex et al. 2011, 2012], which parses Haskell to prove specified properties. When a proof is stuck, Zeno conjectures and tries to proves lemmas. For conjecturing lemmas, Zeno follows the generalization technique commonly found in Boyer-Moore inductive theorem provers [Boyer et al. 1995]. Generalization transforms an equality with a repeated subexpression into an equality with the subexpression replaced by a variable. The transformed equality is posited as a conjecture which Zeno proves on the side. To avoid over-generalization and quickly filter out false lemmas, Zeno searches for counterexamples before attempting a proof.

HipSpec [Claessen et al. 2012], built on QuickSpec [Claessen et al. 2010] and Hipster [Valbuena and Johansson 2015], is another inductive theorem prover for Haskell. Unlike Zeno, HipSpec does not use generalization to conjecture lemmas during a proof. Instead, it compiles a library of proven lemmas that may be useful for proofs. To this end, HipSpec generates all possible Haskell expressions that state equalities up to some depth. These generated equalities are postulated and checked before any proof is attempted by passing them through two filtering steps: (1) QuickCheck [Claessen and Hughes 2000] to find counterexamples and (2) the Z3 SMT solver [de Moura and Bjørner 2008] to prove equalities. The remaining expressions are fed into HipSpec's internal automatic inductive theorem prover. HipSpec has been used to prove that common implementations of certain type classes satisfy the type class' laws [Arvidsson et al. 2019].

TheSy [Singher and Itzhaky 2021] uses equality-graphs (or Program Expression Graphs) to compactly represent equalities. The technique enables efficient selection of a canonical representative of an equivalent class of programs. Like HipSpec, TheSy enumerates potential lemmas that may be helpful for a proof. The equality-graph allows TheSy to improve the enumeration of lemmas by filtering out (and effectively removing the need to re-attempt proofs for) equivalent programs.

MATHsAiD [McCasland et al. 2017] is another tool in the style of HipSpec which assists mathematicians in exploring theories. Yang et al. [2019] present a theorem prover that uses techniques from program synthesis to guide its lemma generation.

Propel's lemma generation is closest to HipSpec's and TheSy's as it enumerates upfront a list of lemmas that may be useful. While both HipSpec and Propel scan the environment for symbols for lemma exploration, Propel restricts itself to each function's local context and relevant data constructors. Propel might miss some potentially useful lemmas involving two distantly-related functions but completes proofs faster and – for the properties in which we are interested – more effectively than HipSpec as our evaluation shows. Further, while HipSpec posits lemmas before any proof attempt, Propel interleaves this process with each function, so lemmas about future functions are never conjectured before the proofs for the current function are complete. We believe a combination of TheSy's equality-graph representation and Propel's strategy to conjecture lemmas for proving algebraic properties has the potential to lead to further improvements.

Proof by Consistency. Inductive theorem provers labeled under *inductionless induction* [Comon 2001] or *proof by consistency* [Wirth 2005] treat induction implicitly. This treatment has been revived in the cyclic proof theory [Brotherston 2005; Sprenger and Dam 2003], e.g., in the generic automated theorem prover Cyclist [Brotherston et al. 2012]. An extension to GHC, CycleQ [Jones et al. 2022], contains a theorem prover to help in equational reasoning. Reynolds and Kuncak [2015] added support for induction to the CVC4 [Barrett et al. 2011] SMT solver by skolemizing the inductive hypothesis. The SMT solver finds a model that satisfies the negation of the induction hypothesis and fails if the theorem is true. All these systems treat properties as assertions and do not track them at the type level, unlike Propel. Tracking properties in the types allows higher-order functions to enforce properties about their inputs that are checked at the call site.

Refinement Types and LiquidHaskell. Refinement types [Rushby et al. 1998; Xi and Pfenning 1998] as implemented in LiquidHaskell (LH) [Vazou et al. 2014] allow programmers to attach propositions to data types to refine the domain of the type. These systems are the closest to our approach in that they also capture additional properties at the type level. Whereas LH offloads the checking of the refinement predicates attached to types to an SMT solver, LiquidHaskell with Refinement Reflection (LH+RR) [Vazou et al. 2017] enables programmers to prove propositions by constructing proof objects that witness that the propositions hold – either manually or by invoking the *Proof by Logical Evaluation (PLE)* proof-search algorithm (which can automate certain proofs but requires developers to provide the structure of the induction by specifying the arguments to the recursive call). A major limitation of LH and LH+RR is that no quantifiers are allowed in refinements, prohibiting refinement types such as the one expressing that `zipWith` is commutative if it is given a commutative function, i.e., the type `{ f:a → a → b | forall (x:a, y:a), f a b = f b a }` `→ { g:[a] → [a] → [b] | forall (x:[a], y:[a]), g x y = g y x }` for `zipWith` is not valid in LH. Propel allows expressing this type as shown in Listing 2.

A way to represent, a commutative function in LH+RR is by pairing the function definition with its proof of commutativity. Such proof values must be passed manually, incurring significant overhead, even if PLE derives the proof automatically – which is rare for the CRDTs in our evaluation.

Correct Algebraic Properties. Servois [Bansal et al. 2018] generates conditions, by iteratively refining a starting condition, under which two functions commute. Thus two functions commute

if their commutativity condition is always true. All these approaches study the commutativity between functions rather than the commutativity of a binary function's arguments, thus they are more suited for applications to parallel programming [Pottenger 1998]. Aleen and Clark [2009] build a static analysis technique that probabilistically determines whether an imperative function commutes with itself based on how it modifies a random memory layout. Instead of verifying that a function is commutative, Gélineau [2010] design a Haskell library where commutative functions are so by construction. A commutative function is a computation wrapped in a monad that imposes an ordering on the values applied to it. Hence the computation cannot observe their original order. However, it is not clear how to define associative functions by construction. Liu et al. [2020] extend LiquidHaskell [Vazou et al. 2014] and Refinement Reflection [Vazou et al. 2017] to attach laws to type class definitions – expressed as members of the type class that instances must implement. Propel is different than type class refinements (TCR) in two ways: First, Propel restricts itself to predefined algebraic properties which gives rise to higher level of automation. Second, making a function an instance of type class (e.g., `Commutativity`) requires expressing the function as data type. Hence, to assert algebraic properties of functions in TCR, functions must be defunctionalized [Reynolds 1972], higher-order functions become type functions parametrized by the defunctionalization identifiers, and functions' closures must be handled manually. No such transformation is needed in Propel thanks to expressing algebraic properties directly in a function's type.

Expressive Type Systems. Dependent type systems such as Coq's calculus of constructions [Coquand and Huet 1986] or Agda's type system [Bove et al. 2009] can encode functional properties by lifting a proof term to the type level. These proof terms must be manually constructed by the user.

Type systems that track properties through type annotations have been explored in multiple domains. For example, information flow type systems [Heintze and Riecke 1998; Sabelfeld and Myers 2003] trace the flow of private information through a program. Cortier et al. [2017] the authors present a type system that guarantees privacy properties of protocols.

8 CONCLUSION

CRDTs ensure that replicated data in a distributed system eventually converge to the same state for each replica. To guarantee this behavior, the operations on the CRDT must obey certain algebraic properties (i.e., commutativity, associativity, idempotence). As a result, developers need to make sure that their implementation adheres to such properties, but there is no machine-checked mechanism to confirm that this is the case. Hence, developers usually rely on approaches such as testing, model checking, and theorem proving, which do not cover the complete input space, operate on a model instead of on the program directly, or require manual developer intervention to provide proofs.

In this paper, we proposed Propel, a programming language with a type system that enforces the algebraic properties required by CRDTs. Propel proves such properties automatically and directly on the implementation. In Propel, developers specify that a function obeys a certain property (e.g., commutativity) in the type, and the type system certifies that this is the case. We evaluated Propel on a number of CRDT implementations and showed that our approach outperforms existing tools, which aim to check similar properties on source code, in terms of the number of properties proven in a reasonable amount of time.

ACKNOWLEDGMENTS

The authors wish to thank the anonymous reviewers of PLDI 2023 and Viktor Kunčak for their valuable comments and the engaging discussions that improved the paper. This work is supported by the Swiss National Science Foundation (SNSF), grant 200429, and the Basic Research Fund of the University of St. Gallen (GFF) through the International Postdoctoral Fellowship (IPF) 1031569.

ARTIFACT AVAILABILITY

The artifact is available on Zenodo [Zakhour et al. 2023]. It includes the implementation of Propel as discussed in Section 4 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 5 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 benchmark results.

REFERENCES

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

Paulo Sérgio Almeida, Ali Shoker, and Carlos Baquero. 2018. Delta State Replicated Data Types. *J. Parallel and Distrib. Comput.* 111 (2018), 162–173. https://doi.org/10.1016/j.jpdc.2017.08.003

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

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

Carlos Baquero. 2014. Reference implementations of state-based CRDTs that offer deltas for all mutations. https://github.com/CBaquero/delta-enabled-crdts. Last accessed on 6 Apr 2023.

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

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

Gilles Barthe, Julien Forest, David Pichardie, and Vlad Rusu. 2006. Defining and Reasoning About Recursive Functions: A Practical Tool for the Coq Proof Assistant. In *Functional and Logic Programming*, Masami Hagiya and Philip Wadler (Eds.). Springer-Verlag, Berlin/Heidelberg, Germany, 114–129. https://doi.org/10.1007/11737414_9

Gilles Barthe, Maria Joao Frade, Eduardo Giménez, Luis Pinto, and Tarmo Uustalu. 2004. Type-Based Termination of Recursive Definitions. *Mathematical Structures in Computer Science* 14, 1 (Feb. 2004), 97–141. https://doi.org/10.1017/S0960129503004122

Annette Bieniusa, Marek Zawirski, Nuno Preguiça, Marc Shapiro, Carlos Baquero, Valter Balegas, and Sérgio Duarte. 2012. An Optimized Conflict-free Replicated Set. arXiv:1210.3368

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

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

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

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

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

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

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

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

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

Thierry Coquand and Gérard Huet. 1986. The Calculus of Constructions.

Véronique Cortier, Niklas Grimm, Joseph Lallemand, and Matteo Maffei. 2017. A Type System for Privacy Properties. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (Dallas, Texas, USA) *(CCS '17)*. ACM, New York, NY, USA, 409–423. https://doi.org/10.1145/3133956.3133998

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

C. A. Ellis and S. J. Gibbs. 1989. Concurrency Control in Groupware Systems. In *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data* (Portland, Oregon, USA) *(SIGMOD '89)*. ACM, New York, NY, USA, 399–407. https://doi.org/10.1145/67544.66963

Eduarde Giménez. 1995. Codifying Guarded Definitions with Recursive Schemes. In *Types for Proofs and Programs*, Peter Dybjer, Bengt Nordström, and Jan Smith (Eds.). Springer-Verlag, Berlin/Heidelberg, Germany, 39–59. https://doi.org/10.1007/3-540-60579-7_3

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

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

Michael Heinrichs. 2017. Experimental CRDT-implementations for the JVM. https://github.com/netopyr/wurmloch-crdt. Last accessed on 6 Apr 2023.

Nevin Heintze and Jon G. Riecke. 1998. The SLam Calculus: Programming with Secrecy and Integrity. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Diego, California, USA) *(POPL '98)*. ACM, New York, NY, USA, 365–377. https://doi.org/10.1145/268946.268976

Paul R. Johnson and Robert H. Thomas. 1975. The Maintenance of Duplicate Databases. Internet Requests for Comments. https://www.rfc-editor.org/rfc/rfc677.txt

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

Martin Kleppmann. 2017. Automerge. https://github.com/automerge/automerge. Last accessed on 6 Apr 2023.

Martin Kleppmann and Alastair R. Beresford. 2017. A Conflict-Free Replicated JSON Datatype. *IEEE Transactions on Parallel and Distributed Systems* 28, 10 (2017), 2733–2746. https://doi.org/10.1109/TPDS.2017.2697382

Shadaj Laddad, Conor Power, Mae Milano, Alvin Cheung, and Joseph M. Hellerstein. 2022. Katara: Synthesizing CRDTs with Verified Lifting. *Proceedings of the ACM on Programming Languages* 6, OOPSLA2, Article 173 (Oct. 2022), 29 pages. https://doi.org/10.1145/3563336

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

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

Christopher Meiklejohn. 2016. Prototype implementation of Conflict-free Replicated Data Types (CRDTs) in Erlang. https://github.com/lasp-lang/types. Last accessed on 6 Apr 2023.

Kartik Nagar and Suresh Jagannathan. 2019. Automated Parameterized Verification of CRDTs. In *Computer Aided Verification*, Isil Dillig and Serdar Tasiran (Eds.). Springer International Publishing, Cham, 459–477. https://doi.org/10.1007/978-3-030-25543-5_26

Sreeja S. Nair, Gustavo Petri, and Marc Shapiro. 2020. Proving the Safety of Highly-Available Distributed Objects. In *Programming Languages and Systems*, Peter Müller (Ed.). Springer International Publishing, Cham, 544–571. https://doi.org/10.1007/978-3-030-44914-8_20

Petru Nicolaescu, Kevin Jahns, Michael Derntl, and Ralf Klamma. 2015. Yjs: A Framework for Near Real-Time P2P Shared Editing on Arbitrary Data Types. In *Engineering the Web in the Big Data Era*, Philipp Cimiano, Flavius Frasincar, Geert-Jan Houben, and Daniel Schwabe (Eds.). Springer International Publishing, Cham, 675–678. https://doi.org/10.1007/978-3-319-19890-3_55

Tobias Nipkow, Markus Wenzel, and Lawrence C Paulson. 2002. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. Springer-Verlag, Berlin/Heidelberg, Germany. https://doi.org/10.1007/3-540-45949-9

William M. Pottenger. 1998. The Role of Associativity and Commutativity in the Detection and Transformation of Loop-Level Parallelism. In *Proceedings of the 12th international conference on Supercomputing (ICS '98)*. ACM, New York, NY, USA, 188–195. https://doi.org/10.1145/277830.277870

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

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

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

David Rusu. 2016. A collection of well-tested, serializable CRDTs for Rust. https://github.com/rust-crdt/rust-crdt. Last accessed on 6 Apr 2023.

A. Sabelfeld and A.C. Myers. 2003. Language-Based Information-Flow Security. *IEEE Journal on Selected Areas in Communications* 21, 1 (2003), 5–19. https://doi.org/10.1109/JSAC.2002.806121

Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011a. A Comprehensive Study of Convergent and Commutative Replicated Data Types. https://hal.inria.fr/inria-00555588

Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011b. 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

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

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

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

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

Bartosz Sypytkowski. 2018. An example implementations of various CRDTs. https://github.com/Horusiath/crdt-examples. Last accessed on 6 Apr 2023.

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

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

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

Werner Vogels. 2009. Eventually Consistent. *Commun. ACM* 52, 1 (Jan. 2009), 40–44. https://doi.org/10.1145/1435417.1435432

Matthew Weidner, Heather Miller, and Christopher Meiklejohn. 2020. Composing and Decomposing Op-Based CRDTs with Semidirect Products. *Proceedings of the ACM on Programming Languages* 4, ICFP, Article 94 (Aug. 2020), 27 pages. https://doi.org/10.1145/3408976

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

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

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

George Zakhour, Pascal Weisenburger, and Guido Salvaneschi. 2023. *Type-Checking CRDT Convergence.* https://doi.org/10.5281/zenodo.7817421

Peter Zeller, Annette Bieniusa, and Arnd Poetzsch-Heffter. 2014. Formal Specification and Verification of CRDTs. In *Formal Techniques for Distributed Objects, Components, and Systems*, Erika Ábrahám and Catuscia Palamidessi (Eds.). Springer-Verlag, Berlin/Heidelberg, Germany, 33–48. https://doi.org/10.1007/978-3-662-43613-4_3

Received 2022-11-10; accepted 2023-03-31