Composing CRDTs Convergent by Construction

ALEXANDER STÄDING DOMINGUEZ✉, GEORGE ZAKHOUR, PASCAL WEISENBURGER, and GUIDO SALVANESCHI, University of St. Gallen, Switzerland

Conflict-Free Replicated Data Types (CRDTs) are abstract data types that ensure eventual convergence among data replicas in distributed systems. As they provide convergence out-of-the-box, CRDTs have become key building blocks for highly available, collaborative, and offline-capable systems, powering applications from real-time editors to distributed databases. Adopting an *individual* CRDT is straightforward, but real-world software routinely requires *composing* them. For example, an application might store a set of counters, combining a set CRDT with a counter CRDT. Unfortunately, classical CRDT theory does not guarantee that a composition of convergent CRDTs converges, forcing developers to reason about convergence again – the very burden CRDTs were introduced to remove.

In this paper, we introduce a compositional framework for a broad class of operation-based CRDTs. It assembles CRDTs from five principal combinators – Product, MapState, Associate, Traverse, and MapInterpretation – each with built-in convergence guarantees. Any CRDT assembled from these combinators is itself a CRDT, preserving convergence *by construction*. This set is free of redundancy and subsumes previously proposed combinators.

We develop the framework, its underlying theory, and its proofs entirely in Lean 4, producing a single artifact that serves as both the formal model and an executable, verified implementation. Our reusable library, Crdtlib, provides implementations and proofs for every combinator and CRDT in this paper. Our case studies (i) implement common CRDTs from Shapiro et al., (ii) apply the combinators in a complete application, and (iii) encode a JSON-structured tree CRDT as expressive as Automerge, with competitive runtime and memory use. These case studies show that developers can compose CRDTs without re-proving convergence for each composite.

CCS Concepts: • **Software and its engineering** → **Data types and structures**; *Software verification*; • **Computing methodologies** → **Distributed algorithms**; • **Theory of computation** → Distributed computing models.

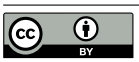
Additional Key Words and Phrases: Conflict-Free Replicated Data Types, Distributed Programming

ACM Reference Format:
Alexander Städing Dominguez, George Zakhour, Pascal Weisenburger, and Guido Salvaneschi. 2026. Composing CRDTs Convergent by Construction. *Proc. ACM Program. Lang.* 10, OOPSLA2, Article 346 (October 2026), 28 pages. https://doi.org/10.1145/3839478

1 Introduction

Replicated-data *consistency* is a major source of complexity in distributed systems [16]. Achieving *strong consistency* requires expensive coordination among replicas, hindering responsiveness and drastically reducing availability. To remain scalable and available during temporary replica outages, many distributed systems adopt *weak consistency* [10], tolerating temporary inconsistencies and propagating changes later.

Authors' Contact Information: Alexander Städing Dominguez, alexander.staedingdominguez@unisg.ch; George Zakhour, george.zakhour@unisg.ch; Pascal Weisenburger, pascal.weisenburger@unisg.ch; Guido Salvaneschi, guido.salvaneschi@unisg.ch, University of St. Gallen, St. Gallen, SG, Switzerland.



This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2026 Copyright held by the owner/author(s).
ACM 2475-1421/2026/10-ART346
https://doi.org/10.1145/3839478

Rather than redesigning consistency logic for each application, researchers have encapsulated it in Abstract Data Types (ADTs) whose replicas *eventually* converge to the same value. CRDTs [41] ensure convergence *by design*, abstracting over the logic that is necessary to achieve eventual convergence. Since their introduction, CRDTs have become an essential tool for building highly available distributed systems that maintain consistency without coordination. Successful applications include the distributed databases Riak [38] and Antidote DB [2], PayPal's mechanism for managing compliance status across data centers [32], Facebook's Apollo low-latency "consistency at scale" database [20], and documents and collaborative editors [3].

Although CRDTs exist for various data structures, designing and verifying complex examples, such as graph-structured data, remains challenging. This complexity stems from operations needing to commute in every concurrent execution, requiring reasoning about ordering, causality, and merge strategies across replicas; designing a new CRDT thus remains a research endeavor [3, 9, 21, 39].

Although an individual CRDT, such as a grow-only set or additive counter, can be adopted *off the shelf* with almost no reasoning, applications rarely manipulate only one abstract data type. They need maps from user IDs to counters, sets of JSON documents, and graph-structured data, naturally calling for the *composition* of existing CRDTs.

Unfortunately, composition can break the algebraic assumptions guaranteeing convergence: operations that commute in isolation can interact subtly once their carriers are composed. For example, if one replica removes a map entry while another concurrently increments its counter, the replicas diverge unless the composite is carefully engineered [1, 37]. Such issues have motivated modular verification of CRDT-based applications, including separately verifying implementations and client programs [31] and decoupling application logic from CRDT correctness proofs [36].

In summary, classical CRDT theory studies each data type in isolation, offering no general condition under which composition preserves convergence. Developers must therefore craft a new monolithic CRDT for every composite, develop complex formal proofs, or risk latent divergence – the very burden CRDTs are meant to remove.

Example. An example of such a composite solution is a list CRDT where elements can be added and removed, i.e., the list supports two operations: *remove(v)* removes the element v from the list, and *add(p, v, n)* adds a new element v (at any position) between p and n. In a sequential setting, such a list can be naturally represented by a linked list. In the presence of concurrent events, however, the internal state of the CRDT cannot be a linked list since concurrent *add* events may result in multiple candidates to be inserted at the same position in the list. Instead, the list is represented internally as a directed acyclic graph (DAG).

In this DAG, inserting an element a between b and c adds edges from b to a and from a to c. An element a cannot simply be removed from the graph because concurrent additions may insert elements before or after it. Instead, the CRDT records removed elements in a "tombstone set." Therefore, our list CRDT is composed of three basic CRDTs: the set of nodes of the DAG, the edges of the DAG, and the tombstone set.

Our solution composes simple CRDTs, such as sets, into complex ones, such as lists, and derives the composite's convergence compositionally from that of its components. To this end, we provide a set of primitive CRDTs and a set of CRDT combinators to build CRDTs that are convergent by construction.

For the list CRDT, the fact that the internal implementation is a DAG should remain opaque to user code, which only interacts with the simpler list interface. Therefore, we must be able to *interpret* the DAG into a list. As there may be multiple candidates for an element at a specific position in the list, we require a "tie-breaker" to pick among the candidates. To respect the "add-between" semantics of the add operation, an element is a candidate for a position in the "interpreted" list if

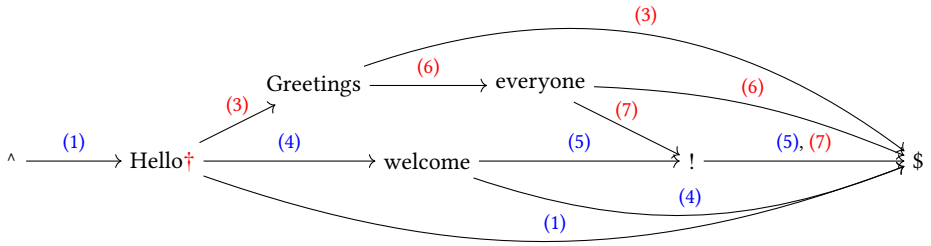


Fig. 1. A list CRDT's internal representation as a DAG.

(1) it has not been removed, (2) it has not been visited for an earlier position in the interpreted list and (3) all its predecessors have already been visited. Our approach guarantees convergence of both the CRDT's composed *internal* state and its *interpretation*.

Figure 1 illustrates this list CRDT over strings, using lexicographic minimum as the tie-breaker. Two users, red and blue, are concurrently editing a text document. The start and end of the list are denoted by ^ and $, respectively. Removed nodes have a † next to them. The operations are processed in this order: (1) add(^, Hello, $), (2) remove(Hello), (3) add(Hello, Greetings, $), (4) add(Hello, welcome, $), (5) add(welcome, !, $), (6) add(Greetings, everyone, $), (7) add(everyone, !, $). Each edge is annotated with the operation that introduced it. Under the lexicographic tie-breaker, the user-observable list evaluates to "Greetings everyone welcome!". We use this list throughout the paper to demonstrate compositional CRDT construction.

Contribution. In this paper, we present a unified basis of five combinators for a broad class of operation-based CRDTs. Compositions built from them converge by construction, as formalized in Theorem 4.1, enabling modular, reusable construction of sophisticated CRDTs from simpler ones.

First, we identify five principal combinators – Product (Section 4.1), MapState (Section 4.2), Associate (Section 4.3), Traverse (Section 4.4), and MapInterpretation (Section 4.5) – that construct basic CRDTs, such as a Two-Phase-Set [40], and complex ones, such as trees, from primitives. While previous work has verified some of these combinators individually [36], we identify them as a unified basis for CRDT composition. Each captures a distinct axis of composition; removing any one makes some construction evaluated here unreachable. We discuss this choice of basis in Section 8.

Second, we demonstrate this modular approach with Crdtlib, a CRDT library derived solely from these principal combinators. It constructs a tree-structured CRDT akin to the widely used Automerge [3], supporting JSON-like hierarchical data with nested lists, maps, numbers, Booleans, strings, and counters. The library is fully mechanized in Lean 4 [12], ensuring correctness *of the actual implementation*. In summary, this paper makes the following contributions:

- We identify Product, MapState, Associate, Traverse, and MapInterpretation as sufficient principal combinators to reconstruct the common CRDTs evaluated in Section 6.1.
- From them, we build an extensible library of derived combinators that incrementally compose increasingly complex CRDTs.
- We formalize the principal combinators and prove that they construct convergent CRDTs, so derived combinators retain convergence *by construction*.
- We compositionally reconstruct common CRDTs from the comprehensive study of Shapiro et al. [40].
- We evaluate the combinators' ergonomics in a case study by constructing even highly complex CRDTs, such as the Automerge [3] tree, from them.

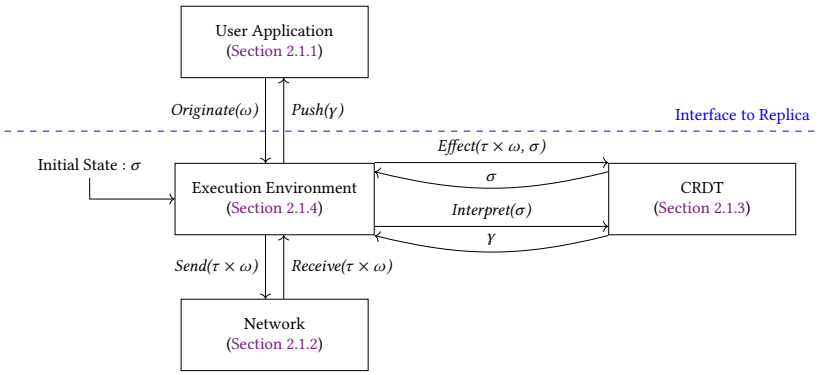


Fig. 2. Overview of the System Model.

2 Operation-Based Conflict-Free Replicated Data Types

This section defines our system model and CRDTs. We first recall Eventual Consistency (EC) and Strong Eventual Consistency (SEC) [41], the central guarantees of CRDTs:

Definition 2.1 (Eventual Consistency (EC)). A system is *eventually consistent* if it enjoys these properties: (1) *eventual delivery*: every update delivered at some correct replica is eventually delivered to all correct replicas, (2) *convergence*: all correct replicas that receive the same updates eventually reach equivalent states, and (3) *termination*: all method executions terminate.

Definition 2.2 (Strong Eventual Consistency (SEC)). A system is *strongly eventually consistent* if it is *eventually consistent* (Definition 2.1) and *strongly convergent*: all correct replicas that receive the same updates have equivalent states.

Informally, with plain EC, convergence is only a long-term liveness property. SEC additionally provides a safety property: replicas cannot disagree after seeing the same updates, so no later update must repair their values.

The defining property of CRDTs is achieving SEC in a distributed setting without coordination or consensus. Operation-based CRDTs achieve this by propagating state-changing *operations* between replicas [41]. Metadata, commonly a partially ordered logical timestamp [27], records whether one operation *happened before* another or whether they are *concurrent*. Replicas apply operations according to this order. Assuming exactly-once application, concurrent operations must commute for replicas to converge [40]. Our guarantees are scoped to operation-based replication under the delivery assumptions we state in Section 2.1.

2.1 System Model

In our model (Figure 2), each replica is a node in an asynchronous network. Replicas hold strongly eventually consistent copies of a mutable value and exchange messages with other replicas. We assume three delivery properties on the network:

- Messages are delivered in *causal order*. Specifically, a message is not received before all of its causal predecessors are received.
- Every message delivered to one replica is eventually delivered to all replicas, known as *eventual delivery* (Definition 2.1).
- Every message is delivered to each replica *exactly once*, so an event takes effect at most once per replica.

Operations *originate* from a local user or application and then take *effect* locally. An *event* packages an operation with its logical timestamp (Definition 2.3), which advances locally after each local operation. Two replicas may advance logical time independently, producing independent timestamps. A vector clock is an example of a logical timestamp [15, 34] that defines a partial ordering on events. Locally originated events are *sent* over the network and eventually *received* by every replica. We assume no Byzantine faults.

Definition 2.3 (Event). An event is a tuple $\tau \times \omega$ consisting of a timestamp and an operation.

In Figure 2, *User Application* is the application layer, and *Network* abstracts message exchange. *CRDT* encapsulates the effect and interpretation functions that update and read state (Section 2.2). *Execution Environment* is the local runtime of each replica, which maintains the state and connects the components. The four components interact through the following procedure calls.

2.1.1 User Application: Originate and Push. The application invokes *Originate(ω)* on the execution environment for every local operation ω and the execution environment invokes *Push(γ)* on the application with an "interpreted" state γ whenever the CRDT's state σ is updated. The interpreted state for use in an application can differ from the state stored inside the CRDT, from which the interpreted state is derived. For example, a CRDT might store the set of all concurrent events it received but conceptually only takes one of these concurrent events into account according to some deterministic arbitration function (e.g., Last Writer Wins [11]).

2.1.2 Network: Send and Receive. The network layer invokes *Receive($\tau \times \omega$)* on the execution environment in causal order and the execution environment invokes *Send($\tau \times \omega$)* on the network layer to broadcast a new local event $\tau \times \omega$ to other replicas.

2.1.3 CRDT: Effect and Interpretation. An operation-based CRDT consists of two functions: (1) the *effect function* (Definition 2.4), which transforms the state σ into σ' given an event comprising a timestamp τ and an operation ω [5]; and (2) the *interpretation function* (Definition 2.5), which exposes internal state to the application. The execution environment invokes *Effect($\tau \times \omega$, σ)* on the CRDT when a new event $\tau \times \omega$ is received, which returns the CRDT's updated state. The execution environment further invokes *Interpret(σ)* on the CRDT to interpret the CRDT's state σ, which returns the "interpreted" state to be provided to the application layer.

Definition 2.4 (Effect Func.). The effect function $\phi : \tau \times \omega \to \sigma \to \sigma$ computes the next state for a CRDT.

Definition 2.5 (Interpretation Func.). The interpretation function $\psi : \sigma \to \gamma$ computes the interpreted state for a CRDT.

2.1.4 Execution Environment. Each replica has a local runtime for every CRDT, managing interactions among the CRDT, the application, and the network (Algorithm 1). This local execution environment is parameterized

Algorithm 1 Local Execution Environment

```
1:  input: crdt : (φ, ψ)        ▷ Effect and Interpretation functions
2:  input: s_0 : σ                                  ▷ Initial State
3:
4:  s : σ ← s_0                                     ▷ Local State
5:  t : τ ← t_0                                     ▷ Local Timestamp
6:
7:  procedure ORIGINATE(o : ω)                      ▷ Event originated by user
8:      t ← INCREMENT(t)                            ▷ Increment timestamp
9:      e ← (t × o)                                 ▷ Package event
10:     s ← EFFECT(e, s)                            ▷ Apply local event to local state
11:     g ← INTERPRET(s)                            ▷ Compute new external state
12:     PUSH(g)                                     ▷ Push external state to application
13:     SEND(e)                                     ▷ Broadcast event to peers
14:
15: procedure RECEIVE(e : τ × ω)                    ▷ Event received from peer
16:     t ← t ∪ π_τ(e)                              ▷ Merge incoming timestamp
17:     s ← EFFECT(e, s)                            ▷ Apply remote event to local state
18:     g ← INTERPRET(s)                            ▷ Compute new external state
19:     PUSH(g)                                     ▷ Push external state to application
20:
21: procedure EFFECT(e : τ × ω, s : σ) return π_φ(crdt)(e, s)
22: procedure INTERPRET(s : σ) return π_ψ(crdt)(s)
```

by the CRDT and its initial state. For a given CRDT instance, all replicas begin with equivalent initial states. First, the local state and timestamp are assigned their respective initial values (Lines 4 to 5). Two procedures introduce events into the execution environment, and we assume they each execute atomically.

Originate (Line 7) is invoked by a local application to modify the CRDT value (cf. Section 2.1.1). The timestamp t is incremented to reflect a new logical step (Line 8) and is packaged with the operation into an event e (Line 9), which is applied to the local state and interpreted (Lines 10 to 11). The interpreted external state is pushed back to the application (Line 12). Finally, the network sends the event to every replica (Line 13).

Receive (Line 15) handles events received from remote nodes via the network interface (cf. Section 2.1.2). The local timestamp is updated via supremum to reflect the incoming timestamp (Line 16). Like local events, remote events are applied, interpreted, and pushed to the local application (Lines 17 to 19). Effect (Line 21) and Interpret (Line 22) compute the CRDT's updated and interpreted state (cf. Section 2.1.3).

2.2 CRDT Definition

We present our CRDT design in Lean [12], which serves both as an executable implementation and a basis to prove the correctness of the combinators.

SEC is achieved through commutativity of the effect function for concurrent events [41, 46]. Given the partial order $\preceq$ in which each replica receives events, the network may deliver concurrent events in either order.

Definition 2.6 (Concurrency). Events e_1 and e_2 are concurrent under the partial order $\preceq$, written $e_1 ||_{\preceq} e_2$, iff $\neg(e_1 \preceq e_2) \land \neg(e_2 \preceq e_1)$.

The effect function ϕ must therefore produce the same result in either order. In Sections 3 and 4, we prove the following commutativity property for every CRDT effect ϕ (Definition 2.4) and causally unrelated events e_1 and e_2 under $\preceq$.

Definition 2.7 (Effect Function Commutativity). An effect function ϕ is commutative iff

$$\forall e_1, e_2,\ e_1 ||_{\preceq} e_2 \implies \phi(e_1) \circ \phi(e_2) = \phi(e_2) \circ \phi(e_1)$$

The partial order $\preceq$ represents causality. An effect ϕ that commutes under an empty $\preceq$ commutes for all operations. We can now define a CRDT formally.

Definition 2.8 (Conflict-Free Replicated Data Type (CRDT)). A CRDT is a pair $(\phi, \psi)_{\preceq}$ of an effect function ϕ and an interpretation function ψ under a partial order $\preceq$, such that ϕ commutes (Definition 2.7) for concurrent events under $\preceq$.

As our CRDT library is implemented in Lean, we use Lean's theorem-proving features to prove this property directly on the executable implementation of our CRDTs.

Event is a tuple $\tau \times \omega$ (Definition 2.3). For an Event e, the projections $e.\mathrm{t}$ and $e.\mathrm{o}$ return the timestamp and operation, respectively.

```
structure Event τ ω where (t : τ) (o : ω)
```

Concurrent states that neither timestamp precedes the other (Definition 2.6). The [PartialOrder τ] constraint requires τ to implement the PartialOrder type class, enabling the ≤ operator:

```
structure Concurrent [PartialOrder τ] (t₁ t₂ : τ) : Prop where concurrent : ¬ (t₁ ≤ t₂) ∧ ¬ (t₂ ≤ t₁)
```

Two events e_1 and e_2 are thus concurrent precisely when Concurrent e_1.t e_2.t holds. From this predicate, we define a timestamp-based $\mathrm{CRDT}_{\mathrm{t}}$ (indicated by the subscript $_{\mathrm{t}}$), whose commutativity requirement holds only for concurrent events.

```
structure CRDTₜ (τ : Type) [PartialOrder τ] (ω σ γ : Type) where
  effect : Event τ ω → σ → σ
  interpret : σ → γ
  commutative : ∀ (e₁ e₂ : Event τ ω), Concurrent e₁.t e₂.t → (effect e₂) ∘ (effect e₁) = (effect e₁) ∘ (effect e₂)
```

CRDT_t is parameterized by four types: timestamp τ, operation ω, state σ, and interpretation γ. The effect function (ϕ in Definition 2.8) applies an event to a state, transforming the state into a new state; interpret (ψ) converts that state to an interpreted value γ. The commutative property requires concurrent operations to commute under the timestamp order.

We also define CRDT (without subscript $_t$), which commutes under *any* ordering relation. Under a maximally relaxed order, every event pair is concurrent, so we remove Concurrent from the premise and timestamps from the definition:

```
structure CRDT (ω σ γ : Type) where
  effect : ω → σ → σ
  interpret : σ → γ
  commutative : ∀ (e₁ e₂ : ω), (effect e₂) ∘ (effect e₁) = (effect e₁) ∘ (effect e₂)
```

Because this definition commutes under any order, it is stronger than CRDT_t and converts trivially to CRDT_t for a specific order. We use simplified CRDT for effects that commute for all events, and CRDT_t only when event order matters to commutativity. We adopt the convention that CRDTs and combinators with subscript $_t$ are typed for CRDT_t and those without for CRDT. By the above conversion, every CRDT x implies its counterpart x_t. Additionally, we omit uninteresting type classes such as DecidableEq and Hashable for readability.

3 Primitive CRDTs

We define the following set of primitive CRDTs – which are not parameterized by other CRDTs – as a basis of our compositional approach. They can be categorized into two families of primitives. The first, ac_fun, captures any convergence behavior expressible as an associative–commutative merge over state.[1] Such a merge is the simplest and most direct way to obtain a commuting effect for our CRDT structure, and most of our primitives (e.g., union_set, counter, gcounter, lww) are instances of it. The second, mv_reg_t, resolves concurrency by comparing a timestamp stored *separately* from the merge state – behavior that ac_fun cannot express, since an associative–commutative merge has no notion of a timestamp. Thus mv_reg_t is necessary and irreducible to ac_fun.

When composed using the combinators (Sections 4 and 5), the resulting CRDTs are guaranteed to converge, i.e., they retain strong eventual consistency; we omit the commutativity proofs here.

The primitives all use the identity interpretation – more interesting interpretation functions are needed for more complex CRDTs, like the ones presented in Sections 4.5 and 5.4.

Associative and Commutative Function. ac_fun accepts any commutative and associative function f, together with proofs that both properties hold, i.e., f obeys the laws encoded in the Std.Commutative and Std.Associative type classes. The resulting CRDT has the same type α for its operation, state and interpretation. Commutativity of the effect function (Definition 2.7) follows from commutativity and associativity of f, making this a valid CRDT:

```
def ac_fun α (f : α → α → α) [Std.Commutative f] [Std.Associative f] : CRDT α α α := { effect := f, interpret := id, ... }
```

Several common primitives follow directly from ac_fun, including counters. Instantiating f with integer addition yields a counter CRDT that can be both incremented and decremented: each operation carries a delta, where decrements are modeled by negative values. Restricting operations to natural numbers yields a grow-only counter (gcounter), which cannot express a negative delta.

[1] This is a relaxation (no idempotence requirement) of the state-based CRDT from the literature.

```
def counter : CRDT Int Int Int := ac_fun Int (· + ·)        def gcounter : CRDT Nat Nat Nat := ac_fun Nat (· + ·)
```

More generally, join produces a CRDT from any join-semilattice: a type with a Least Upper Bound (LUB) (Mathlib's SemilatticeSup [33]). Because LUB is associative and commutative, join specializes ac_fun and stores only the maximum written value.

```
def join (σ : Type) [SemilatticeSup σ] : CRDT σ σ σ := ac_fun σ max
```

Many CRDTs instantiate join. One of the simplest is union_set, a grow-only set (G-Set) variant whose elements can be added but never removed. Its state evolves by joining over finite sets ordered by inclusion, so the LUB is set union. We use Mathlib 4's finite Finset [33] rather than Set because its membership is decidable by default and its full state inspection always terminates.

```
def union_set (σ : Type) := join (Finset σ)
```

Another instance is the Last Writer Wins (LWW) register, which pairs each value with a logical timestamp in a Lamport tuple and orders pairs lexicographically: first by timestamp, then by value to break ties. Since this ordering is total, it forms a join-semilattice whose LUB retains the value with the greatest timestamp, giving LWW semantics:

```
structure Lamport (σ : Type) where (t : Nat) (s : σ)
instance [LinearOrder σ] : LinearOrder (Lamport σ) := LinearOrder.lift' (λ x ↦ toLex (x.t, x.s)) (by ...)
def lww (σ : Type) [LinearOrder σ] := join (Lamport σ)
```

Multi-Value Register. The mv_reg CRDT stores the latest concurrent values and depends on the causal event order from Section 2.1. When an event takes effect, any value carried by an event with a strictly smaller timestamp (x.t < e.t) is removed:

```
def mv_reg_t (σ : Type) [PartialOrder τ] [LinearOrder σ] : CRDT_t τ σ (Finset (Event τ σ)) (Finset (Event τ σ)) where
  effect e s := insert e (Finset.filter (λ x ↦ ¬ x.t < e.t) s)
  interpret := id
```

An example of a CRDT building on mv_reg_t is the mv_lww_t register, which is formally introduced in Section 4.5. Rather than exposing the full set of concurrent values, it breaks ties among them using a total order on values, so that the state is always read as a single value.

4 Principal Combinators

This section introduces our combinators, which are the building blocks for composing operation-based CRDTs from the *primitive CRDTs* introduced in Section 3. A combinator is a function that accepts one or more CRDTs and produces another CRDT. Because the combinator's arguments are provably convergent CRDTs and the combinator is proven correct, the result is also a convergent CRDT, thus enabling compositional reasoning. Therefore, complex CRDTs can be decomposed into simpler ones that each focus on a single aspect of the complete CRDT and then composed together using the proposed combinators.

For readability, we define each principal combinator in terms of the simplified CRDT (Section 2.2). We did, however, implement each principal combinator for both CRDT and CRDT_t. We present our combinators in two ways: using Lean code and pictorially.

Compositional Guarantee. The central guarantee of the framework is compositional:

Theorem 4.1 (Compositional Convergence). *Every well-typed expression built from CRDTs – in particular the primitive CRDTs constructed under the algebraic premises of Section 3 – using the Product, MapState, Associate, Traverse, and MapInterpretation combinators of this section is itself a CRDT (Definition 2.8): its effect function commutes for all operations under CRDT and for all concurrent events under CRDT_t.*

No separate induction theorem is needed as commutativity is a field of the CRDT and CRDT_t structures (Section 2.2). Thus, every well-typed expression carries its own proof and each combinator constructs this field from its arguments. As these commutativity-preservation proofs are mechanized in Lean, composition needs no separate proof beyond the combinator premises. Only map_state introduces an additional algebraic proof obligation (Section 4.2), usually discharged automatically.

For a terminating system with equivalent initial states and the delivery properties of Section 2.1, an effect function with the corresponding commutativity property yields SEC (Definition 2.2) – the standard result for operation-based CRDTs [41]. What Theorem 4.1 adds is that composition preserves the property.

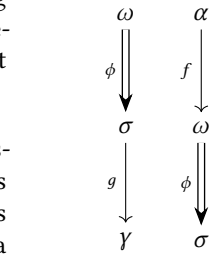


Fig. 3. Effect Composition.

Notation. We borrow notation from commutative diagrams to illustrate CRDT constructions (Figure 3). Single arrows denote functions $\alpha \to \beta$, double arrows denote effects $\alpha \to \beta \to \beta$, and dashed arrows denote the composition produced by a combinator. A CRDT (ϕ, ψ) is a sequence consisting of an effect arrow ϕ and an interpretation arrow.

4.1 Product

This combinator composes two CRDTs such that a single operation is applied to both CRDTs. Figure 4 depicts how operations for the resulting CRDT act on the individual states. The combinator takes two CRDTs (ϕ_1, ψ_1) and (ϕ_2, ψ_2) such that the operation type ω for ϕ_1 and ϕ_2 is the same. The derived CRDT (ϕ', ψ') accepts the same operation type ω but maintains a state consisting of a tuple $\sigma_1 \times \sigma_2$. The effect ϕ' applies operations ω simultaneously to both substates.

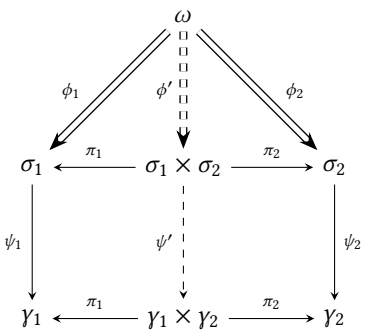


Fig. 4. Product.

Example. In the running example of the list CRDT, adding a new element to the list between two existing elements requires modifying both the set of vertices and the edges of the graph. The product is the building block that allows an operation to affect multiple substates of a CRDT. The running example uses the product combinator through the joint_product combinator, which is based on product (Section 5.3).

Another example is a CRDT that maintains both a (grow-only) set of strings and the most recent insert into this set. To achieve this result, we combine a gset_t CRDT (Section 6.1) with a mv_lww_t CRDT (Section 4.5):

```
product_t (gset_t String) (mv_lww_t String)
```

The design of the product combinator allows one to derive products of an arbitrary number of components through nesting. For example, it is possible to define a derived product3 combinator in terms of product, which can be used to combine three CRDTs:

```
def product3 (c₁ : CRDT ω σ₁ γ₁) (c₂ : CRDT ω σ₂ γ₂) (c₃ : CRDT ω σ₃ γ₃) : CRDT ω ((σ₁ × σ₂) × σ₃) ((γ₁ × γ₂) × γ₃) :=
  product (product c₁ c₂) c₃
```

Note that, instead of defining a product3 of type $(\sigma_1 \times \sigma_2) \times \sigma_3$, we could analogously define the isomorphic product3′ of type $\sigma_1 \times (\sigma_2 \times \sigma_3)$. Both CRDTs can be converted into one another using the map_state combinator (Section 4.2).

Implementation. In our library, the combinator is implemented as in the following code snippet, taking two CRDTs c_1 and c_2 and producing a CRDT with the same operation type as its inputs and a state type that is a pair of the input state types:

```
def product (c₁ : CRDT ω σ₁ γ₁) (c₂ : CRDT ω σ₂ γ₂) : CRDT ω (σ₁ × σ₂) (γ₁ × γ₂) where
  effect e s := ⟨c₁.effect e s.1, c₂.effect e s.2⟩
  interpret s := ⟨c₁.interpret s.1, c₂.interpret s.2⟩
```

4.2 Map State

The *map state* combinator transforms the state of a CRDT, provided that the transformation admits a left inverse. The transformation is given by a function f together with a left inverse f'.

Figure 5 shows how, given an effect $\phi : \omega \to \sigma_1 \to \sigma_1$, an interpretation $\psi : \sigma_1 \to \gamma$, and functions $f : \sigma_1 \to \sigma_2$ and $f' : \sigma_2 \to \sigma_1$ so that $f' \circ f = id$, map_state constructs a CRDT with a transformed effect function $\phi' : \omega \to \sigma_2 \to \sigma_2$ and transformed interpretation $\psi' : \sigma_2 \to \gamma$.

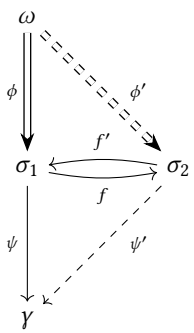


Fig. 5. Map State.

Example. The *map state* combinator is useful to relayout a CRDT's state. For example, if a CRDT is constructed with product3 (Section 4.1) but its state is expected to conform to the one of product3′ (Section 4.1), the first CRDT can be transformed into the second with the following function:

```
def prod3_to_prod3' : CRDT ω ((σ₁ × σ₂) × σ₃) γ → CRDT ω (σ₁ × (σ₂ × σ₃)) γ :=
  map_state (λ ⟨⟨s₁, s₂⟩, s₃⟩ ↦ ⟨s₁, ⟨s₂, s₃⟩⟩) (λ ⟨s₁, ⟨s₂, s₃⟩⟩ ↦ ⟨⟨s₁, s₂⟩, s₃⟩)
```

Implementation. The implementation of this combinator in Crdtlib applies a function f′ to the state before it is passed to the CRDT c, and afterwards applies the function f to the result. Thus, the constructed effect function delegates to the effect c.effect by first mapping via f′ and then mapping back via f. This is why map_state requires a proof object that f' is a left inverse of f, i.e., f′ ∘ f = id (which can often be deduced via Lean's funext and simp tactics):

```
def map_state (f : σ₁ → σ₂) (f' : σ₂ → σ₁) (i : f' ∘ f = id := by funext; simp) (c : CRDT ω σ₁ γ) : CRDT ω σ₂ γ where
  effect e := f ∘ (c.effect e) ∘ f'
  interpret := c.interpret ∘ f'
```

4.3 Associate

The associate combinator constructs CRDTs that are maps. The keys of the map are immutable values and the values are CRDTs. The associate combinator is well-suited for defining trees in the form of path-to-node associations.

Figure 6 depicts how operations on the resulting CRDT affect the states of the CRDT map values. The associate combinator takes a single CRDT (ϕ, ψ) that accepts operations of type ω and captures a state σ. The resulting CRDT produces a map that associates keys κ with instances of σ, i.e., each entry in the map is a CRDT itself, all of the same type. For an effect ϕ that accepts operations ω, the effect ϕ' accepts operations of type $\kappa \times \omega$. For some CRDTs, such as associate, we abbreviate the state type to make the corresponding combinator more evident. The interpretation is a record of two views on the state: map exposes a total view over the key–value space, using the default 0 for undefined keys, and toList enumerates all explicitly set key–value pairs.

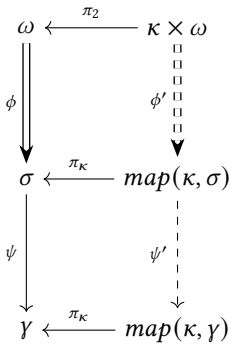


Fig. 6. Associate.

```
abbrev Associate (κ σ : Type) := Std.ExtHashMap κ σ
structure AssociateInterp (κ γ : Type) where (map : κ → γ) (toList [LinearOrder κ] : List (κ × γ))
```

Example. The running example of the list CRDT models the graph's edges as an associate that maps nodes to their set of successors, modeled as a gset (Section 6.1), where each operation adds its successors to the specified node:

```
def edges (σ : Type) : CRDT (σ × σ) (Associate σ (GSet σ)) (AssociateInterp σ (SetInterp σ)) := associate σ (gset σ)
```

Implementation. The effect applies a key–operation event by altering the value at the corresponding key in the hashmap. The delta is calculated using the effect function of the underlying CRDT. If the previous state for that key is undefined, a default value for that state type is used as the base state for the underlying effect function.

```
def associate (κ : Type) [Zero σ] (c : CRDT ω σ γ) : CRDT (κ × ω) (Associate κ σ) (AssociateInterp κ γ) where
  effect e s := s.alter e.fst (λ x ↦ some (c.effect e.snd (x.getD 0)))
  interpret s := { map k := c.interpret (s.getD k 0), toList := Std.ExtHashMap.toListSorted (s.map (λ _ ↦ c.interpret)) }
```

4.4 Traverse

The *traverse* combinator makes it possible to apply multiple operations based on a single operation.

Figure 7 shows how the combinator constructs a CRDT with a transformed effect function $\phi' : \omega_2 \to \sigma \to \sigma$, by applying the function $f : \omega_2 \to [\omega_1]$ to the incoming operation ω_2, transforming the operation into a list of operations of type ω_1. The combinator then applies each operation in the list to the given effect function $\phi : \omega_1 \to \sigma \to \sigma$. The interpretation ψ of the original CRDT does not change.

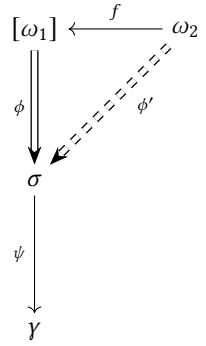


Fig. 7. Traverse.

Example. The running example of the list CRDT supports an operation which inserts an element between existing elements. The ordering of elements is represented by edges in a graph structure. An operation carries three arguments $(prev, elem, next)$ which results in two edges in the graph: from prev to elem and from elem to next. Hence, we use traverse to transform one operation into two edge insertion operations.

We define the edges for the list CRDT's partial order on elements based on the edges CRDT defined in terms of the associate combinator (Section 4.3):

```
def po_edges (σ : Type) : CRDT (σ × σ × σ) (Associate σ (GSet σ)) (AssociateInterp σ (SetInterp σ)) :=
  traverse (λ ⟨prev, elem, next⟩ ↦ [⟨prev, elem⟩, ⟨elem, next⟩]) (edges σ)
```

Implementation. The implementation for the traverse combinator composes c with a fold over the list provided by f, which applies each event in sequence using the effect function:

```
def traverse (f : ω₂ → List ω₁) (c : CRDT ω₁ σ γ) : CRDT ω₂ σ γ where
  effect e := (f e).foldr c.effect
  interpret := c.interpret
```

4.5 Map Interpretation

This combinator transforms a CRDT's interpretation. Figure 8 shows how the derived interpretation function ψ' delegates to the original interpretation ψ and uses f to transform the value from γ_1 to γ_2. The effect function remains the same.

Example. The running example of the list CRDT interprets its graph state – consisting of the set of vertices and the edges between them plus a tombstone set of removed vertices – as a list that respects the partial order induced by the graph structure without the removed elements. Assuming a po_traversal function that performs a traversal on the graph so that its result is such a list, then the list CRDT can be constructed based on a CRDT graph which contains the graph state simply through: map_interpretation po_traversal graph

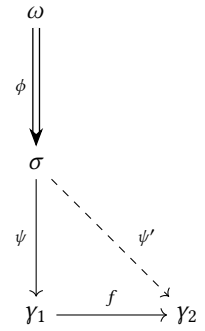


Fig. 8. Map Interpretation.

Implementation. In our library, the combinator is implemented as in the following code snippet, applying a mapping f to the result of the interpretation function of the CRDT c.

```
def map_interpretation (f : γ₁ → γ₂) (c : CRDT ω σ γ₁) : CRDT ω σ γ₂ where
  effect := c.effect
  interpret := f ∘ c.interpret
```

LWW as a Specialization of MV-Reg. A LWW register is a concurrent register that retains only the value(s) written at the latest timestamp according to the partial order on τ, returning a single value by breaking ties among concurrent writes. At first glance one might expect to implement LWW by simply overwriting the stored value whenever a newer timestamp arrives and discarding all others. This is the approach taken by the single-value variant (Section 6.1). That implementation, however, requires a *total* order on timestamps [40] so that the winner can be decided immediately on receipt, with no need to retain any other events.

When only a *partial* order on timestamps is available, concurrent writes cannot be resolved on arrival: an event arriving later may be incomparable to one already stored, making it impossible to discard either without risking data loss. The mv_reg (Section 3) handles this by retaining all mutually incomparable events, filtering out those that are strictly smaller.

The tie-breaker is deferred to the interpretation function. Because mv_reg_t's state is a Finset of all surviving concurrent events, map_interpretation can apply an arbitrary total order on σ to pick one winner from the concurrent set, here via Finset.max:

```
def mv_lww_t (σ : Type) [PartialOrder τ] [LinearOrder σ] [Zero σ] : CRDT_t τ σ (Finset (Event τ σ)) σ :=
  map_interpretation_t (λ s ↦ match (s.image (·.o)).max with | .none => 0 | .some x => x) (mv_reg_t σ)
```

The state type Finset (Event τ σ) is preserved exactly as in mv_reg; only the interpretation of that state changes. This illustrates the central role of map_interpretation: the effect function – and therefore the convergence proof – is inherited from mv_reg, while the choice of a tie-breaker is deferred to the interpretation.

5 Derived Combinators

In this section, we show how to derive combinators from the principal combinators introduced in Section 4. Crucially, derived combinators delegate their proof obligation to the underlying principal combinators, requiring no additional proof effort for CRDT convergence.

5.1 Map Operation

The map_op combinator adapts the operation type of a CRDT by applying a pure function $f : \omega_2 \to \omega_1$ to every incoming operation, translating it into exactly one operation of the underlying type. It is a specialization of traverse (Section 4.4) in which the list-valued function is always List.singleton $\circ f$: the list produced by traverse is guaranteed to be a singleton, so no operation is ever skipped or duplicated – in contrast to traverse's general case, which allows fanout or filtering.

Example. A common use of map_op is to expose a semantically meaningful, user-facing operation type while keeping the internal CRDT machinery primitive. In the running list example, the public interface is given by two operation structs:

```
structure Insert where (prev : σ) (elem : σ) (next : σ)
structure Remove where (elem : σ)
```

Neither struct matches the primitive operation types of the underlying CRDTs: gset expects a bare σ, and po_edges expects a tuple $\sigma \times \sigma \times \sigma$. The map_op combinator bridges this gap by projecting each struct down to the required primitive, leaving the underlying state and interpretation untouched.

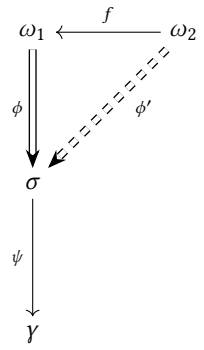


Fig. 9. Map Operation.

For the insert path, the Insert struct is first projected to a $(elem, (prev, elem, next))$ tuple consumed by the joint_product (Section 5.3) of gset and po_edges:

```
(map_op (λ ⟨p, v, n⟩ ↦ ⟨v, ⟨p, v, n⟩⟩) (joint_product (gset σ) (po_edges σ)))
```

For the remove path, Remove carries a single field, which map_op unwraps to recover the bare element expected by the tombstone gset:

```
map_op (λ ⟨o⟩ ↦ o) (gset σ)
```

The two paths are then combined via disjoint_product (Section 5.2), so the complete list accepts Insert σ ⊕ Remove σ as its operation type. map_op thus acts as an adapter layer, decoupling the public interface of a composed CRDT from the operational primitives of its components.

Implementation. Since every incoming operation must produce exactly one translated operation, map_op wraps the result of f in a singleton list and delegates to traverse:

```
def map_op (f : ω₂ → ω₁) (c : CRDT ω₁ σ γ) : CRDT ω₂ σ γ := traverse (List.singleton ∘ f) c
```

The fold inside traverse always receives a one-element list, so its behavior collapses to a single application of c.effect: $\phi'(\omega) = \phi(f(\omega))$. Hence, map_op is strictly less expressive than traverse (cannot express fanout or filtering) but guarantees that every operation is applied exactly once.

5.2 Disjoint Product

The disjoint product combines two CRDTs into a product where each component is updated *independently* via its own operation type – in contrast to the product combinator (Section 4.1), where both components share the same operation type and every operation is applied to both.

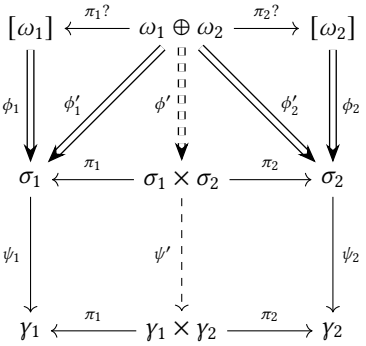


Fig. 10. Disjoint Product.

Figure 10 shows how the combinator is derived from two principal combinators. The inputs are two CRDTs (ϕ_1, ψ_1) and (ϕ_2, ψ_2) with operation types ω_1 and ω_2, respectively. Each is first lifted by traverse (Section 4.4) to accept the tagged union $\omega_1 \oplus \omega_2$: the left traverse extracts ω_1 operations via Sum.getLeft?, yielding a singleton list for left-tagged operations and an empty list – effectively a no-op – for right-tagged ones; the right traverse is analogous. The two resulting CRDTs (ϕ_1', ψ_1) and (ϕ_2', ψ_2) now share the operation type $\omega_1 \oplus \omega_2$, and are therefore directly composable via product (Section 4.1) into the final CRDT (ϕ', ψ') with state $\sigma_1 \times \sigma_2$ and interpretation $\gamma_1 \times \gamma_2$.

Example. Consider a two-dimensional point whose x and y coordinates evolve independently. Modeling each coordinate as an mv_reg CRDT (Section 3) over integers, the point is composed as: disjoint_product$_t$ (mv_reg$_t$ Int) (mv_reg$_t$ Int)

An operation of type Int ⊕ Int then targets either the x-coordinate (Sum.inl) or the y-coordinate (Sum.inr), leaving the other unchanged. A line segment with independent start and end points follows by nesting: disjoint_product point point

Implementation. The implementation applies traverse to each component CRDT, using Option.toList to convert the partial accessors Sum.getLeft? and Sum.getRight? into list-valued functions suitable for traverse: an absent value yields [] (skip) and a present value yields a singleton list (apply). The two traverse-lifted CRDTs are then joined by product:

```
def disjoint_product (c₁ : CRDT ω₁ σ₁ γ₁) (c₂ : CRDT ω₂ σ₂ γ₂) : CRDT (ω₁ ⊕ ω₂) (σ₁ × σ₂) (γ₁ × γ₂)
 := product (traverse (Option.toList ∘ Sum.getLeft?) c₁) (traverse (Option.toList ∘ Sum.getRight?) c₂)
```

5.3 Joint Product

We present a combinator that, like the disjoint product (Section 5.2), combines two CRDTs with distinct operation types. The key distinction is that the composed operation type is a product $\omega_1 \times \omega_2$ rather than a tagged union $\omega_1 \oplus \omega_2$: every operation carries one sub-operation for each component CRDT, and both are always applied *jointly* in a single step – in contrast to the disjoint product where each incoming operation targets *either* ϕ_1 or ϕ_2.

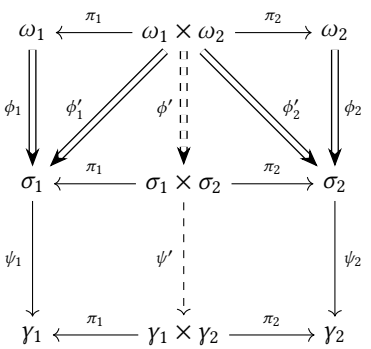


Fig. 11. Joint Product.

Figure 11 shows how the combinator mirrors the structure of disjoint product (Section 5.2), differing only in the choice of operation projections. The inputs are two CRDTs (ϕ_1, ψ_1) and (ϕ_2, ψ_2) with operation types ω_1 and ω_2, respectively. Each is lifted by traverse (Section 4.4) to accept the product type $\omega_1 \times \omega_2$: the left traverse extracts the first component via Prod.fst, and the right traverse extracts the second via Prod.snd. Unlike the partial accessors Sum.getLeft? and Sum.getRight? used in disjoint_product, both projections are total, so every incoming operation is delivered to *both* components without exception. The two lifted CRDTs (ϕ_1', ψ_1) and (ϕ_2', ψ_2) share the operation type $\omega_1 \times \omega_2$ and are combined via product (Section 4.1) into the final CRDT (ϕ', ψ') with state $\sigma_1 \times \sigma_2$ and interpretation $\gamma_1 \times \gamma_2$.

Example. Consider a two-dimensional displacement vector whose x- and y-components are modeled as separate mv_reg CRDTs. Unlike the disjoint point from Section 5.2, where each coordinate is updated independently, a joint product enforces that both coordinates are always moved atomically together: joint_product (mv_reg Int) (mv_reg Int)

A move operation then has type Int × Int, carrying new values for both x and y simultaneously, so no intermediate state exists in which only one coordinate has been updated.

Implementation. The combinator accepts operations of type $\omega_1 \times \omega_2$ and routes the first component to ϕ_1 and the second to ϕ_2. Since both components are always present, no partial unwrapping is needed: Prod.fst and Prod.snd replace the partial accessors Sum.getLeft? and Sum.getRight? used in the disjoint product, and are lifted through the traverse combinator (Section 4.4) as singleton lists, ensuring each sub-operation is delivered exactly once. The two resulting CRDTs are then

joined with the product combinator (Section 4.1), making the overall structure of joint_product and disjoint_product identical up to the choice of operation projections.

```
def joint_product (c₁ : CRDT ω₁ σ₁ γ₁) (c₂ : CRDT ω₂ σ₂ γ₂) : CRDT (ω₁ × ω₂) (σ₁ × σ₂) (γ₁ × γ₂) :=
  product (traverse (List.singleton ∘ Prod.fst) c₁) (traverse (List.singleton ∘ Prod.snd) c₂)
```

5.4 Tagged Union

We present a combinator that constructs a CRDT to represent a tagged union, which (like the disjoint product before) is a fundamental building block for encoding algebraic data types as CRDTs.

Example. For example, a geometric object that can be either a point or a line can be represented as an enumeration of both variants. Assuming the point and line CRDTs presented before, the geometric object is composed as: tagged_union point line

Implementation. The combinator combines two CRDTs and a multi-value register CRDT into a tagged union, where the register holds the Choice tag indicating whether the union represents the Choice.left or Choice.right CRDT. The operation updates either the left or the right CRDT's value or the tag. Thus, the tagged union CRDT always retains the state for both elements, with the tag indicating which element should be retrieved when interpreting the CRDT's state. The implementation uses the disjoint_product combinator (Section 5.2) to compose the left value, the right value, and the tag, and the map_interpretation combinator (Section 4.5) to interpret the state into a tagged union, implemented as tagged_union_interp:

```
inductive Choice
 | left
 | right

def tagged_union_interp : (γ₁ × γ₂) × Choice → γ₁ ⊕ γ₂
 | ⟨⟨s₁, _⟩, Choice.left⟩ => Sum.inl s₁
 | ⟨⟨_, s₂⟩, Choice.right⟩ => Sum.inr s₂
```

```
def tagged_union_t [PartialOrder τ] (c₁ : CRDT_t τ ω₁ σ₁ γ₁) (c₂ : CRDT_t τ ω₂ σ₂ γ₂) :
  CRDT_t τ ((ω₁ ⊕ ω₂) ⊕ Choice) ((σ₁ × σ₂) × (Finset (Event τ Choice))) (γ₁ ⊕ γ₂) :=
 map_interpretation_t tagged_union_interp (disjoint_product_t (disjoint_product_t c₁ c₂) (mv_lww_t Choice))
```

Composite State with MapState. The state produced by applying the combinator mirrors the syntax tree of the composition rather than the structure a developer has in mind. A three-way tagged union by nesting two, (tagged_union$_t$ (tagged_union$_t$ c_1 c_2) c_3), yields the state:

$$\Big(\big(\underbrace{(\sigma_1 \times \sigma_2)}_{c_1,\, c_2} \times \underbrace{\mathsf{Finset}\,(\mathsf{Event}\,\tau\,\mathsf{Choice})}_{\text{inner tag}}\big) \times \underbrace{\sigma_3}_{c_3}\Big) \times \underbrace{\mathsf{Finset}\,(\mathsf{Event}\,\tau\,\mathsf{Choice})}_{\text{outer tag}}$$

The three component states are split across two levels of nesting, and the two tag registers are separated by σ_3. Each additional branch makes this imbalance more pronounced: An n-way union carries its components at $n-1$ different depths. MapState (Figure 5) can be used to remove this clutter. Here we use it to flatten the nested products into a single triple with both tag registers collected on the right:

$$(\sigma_1 \times \sigma_2 \times \sigma_3) \times \mathsf{Finset}\,(\mathsf{Event}\,\tau\,\mathsf{Choice}) \times \mathsf{Finset}\,(\mathsf{Event}\,\tau\,\mathsf{Choice}).$$

The two directions of the isomorphism are pure rearrangements of a tuple, so their composition is definitionally the identity and trivially discharges the proof obligation. The resulting CRDT exposes each component state uniformly as σ_n, independent of how deeply the combinators nested it.

6 Evaluation

To evaluate our compositional approach for safely constructing CRDTs from building blocks that converge by construction, we first show that we can implement common CRDTs from the literature [40] (Section 6.1). Second, we apply our approach to a more comprehensive application that, at its core, relies on a custom eventually consistent data type (Section 6.2). Finally, we demonstrate that we can construct a complex CRDT, such as the state-of-the-art Automerge [3], and compare the performance of our implementation (Section 6.3).

6.1 Common CRDTs

In this section, we show how to build the CRDT registers and sets from the classic work of Shapiro et al. [40] by composition using our framework.

6.1.1 Primitives.

Counters. The counter (positive and negative) and gcounter (non-negative) CRDTs are both implemented using the ac_fun primitive (Section 3).

```
def counter : CRDT Int Int Int := ac_fun Int (· + ·)
def gcounter : CRDT Nat Nat Nat := ac_fun Nat (· + ·)
```

Last-Writer-Wins Register (LWW). The single-value LWW register is implemented using the join specialization of ac_fun (Section 3), given a total order on the value type σ:

```
def lww (σ : Type) [LinearOrder σ] := join σ
```

Alternatively, we apply a generalization of the Lamport-based LWW (Section 3). A total order on τ linearizes values written to the register and forms a join-semilattice whose LUB is the maximum, which instantiates join:

```
structure TotallyOrdered (σ : Type) where (t : Nat) (s : σ)
instance [LinearOrder σ] : LinearOrder (TotallyOrdered σ) := LinearOrder.lift' (λ x => toLex (x.t, x.s)) (by ...)
def lww (σ : Type) [LinearOrder σ] := join (TotallyOrdered σ)
```

Multi-Value Register (MVReg). MVReg offers one operation that updates its value and stores the latest concurrent updates. It is exactly our mv_reg primitive (Section 3).

GSet. The gset CRDT models a grow-only set whose elements can be added but not removed. We build it using join with set union, as in union_set (Section 3). `def gset := join (Finset σ)`

For better performance, we use an alternative based on associate (Section 4.3) that is essentially a HashSet. As in Section 4.3, we abbreviate the GSet CRDT's state, which is a HashMap when unfolded. We additionally expose a simple set interface via map_interpretation (Section 4.5), with a membership predicate and a function to enumerate keys.

```
abbrev GSet (σ : Type) := Associate σ Nat
structure SetInterp (σ : Type) where (mem : σ → Bool) (toList [LinearOrder σ] : List σ)
```

This hash-based GSet pairs each key with a gcounter (Section 3). An element is in the set if its gcounter value is greater than zero. Because this counter cannot decrease, an added element cannot be removed, preserving GSet semantics.

```
def gset (σ : Type) : CRDT σ (GSet σ) (SetInterp σ) :=
  map_interpretation
    (λ s ↦ { mem := (· > 0) ∘ s.map, toList := s.toList.filterMap λ ⟨k, v⟩ ↦ if v > 0 then .some k else .none })
    (map_op (λ x ↦ ⟨x, 1⟩) (associate σ gcounter))
```

6.1.2 Two-Phase-Set (2P-Set). Several add/remove set variants share the same two user-facing operations, so we introduce a common operation type used throughout this section:

```
inductive SetOp (σ : Type) where | add (elem : σ) | remove (elem : σ)
```

A 2P-Set is a variant of a set where elements may be added and removed, but once removed they may never be added again. We implement this using a designated tombstone set: disjoint_product (Section 5.2) combines the live and tombstone sets and map_interpretation (Section 4.5) computes the visible set by subtracting the tombstone from the live set:

```
def tpset (σ : Type) : CRDT (SetOp σ) (GSet σ × GSet σ) (SetInterp σ) :=
 map_interpretation (λ ⟨addSet, removeSet⟩ ↦ {
   mem k := addSet.mem k ∧ ¬ removeSet.mem k, toList := addSet.toList.filter (¬ removeSet.mem ·) })
  (map_op (λ | .add elem => .inl elem | .remove elem => .inr elem) (disjoint_product (gset σ) (gset σ)))
```

An element is in the 2P-Set if it is in the *live* set but not in the *tombstone* set. Once a remove operation is issued, the element enters the tombstone gset, which – like all gsets – grows monotonically and never loses entries. Consequently, a removed element can never be re-added.

6.1.3 U-Set. U-Set behaves like 2P-Set but relies on two global invariants: (1) each element is added at most once, and (2) a remove operation is causally delivered after the corresponding add (causal delivery suffices). These invariants prevent concurrent addition and removal of the same element, making convergence trivial and tombstones redundant: elements are simply never re-added, and remove operations can be applied directly to the live set without risk of conflict. In this sense, U-Set is a CRDT [40].

However, its convergence proof relies on *external* invariants – uniqueness of elements and causal message delivery – properties of the network and application layers, not the CRDT itself. Our combinator model encodes convergence intrinsically, as a proof obligation on the effect function, and therefore cannot derive the U-Set without importing these external assumptions.

OR-Set (Section 6.1.6) covers U-Set's practical uses by generating unique tokens internally at each add. It provides the same causal-removal semantics without external uniqueness, yielding a self-contained CRDT derivable in our model.

6.1.4 LWW-Element-Set. The LWW-Element-Set attaches a timestamp to each element rather than to the set as a whole. Like the 2P-Set, it maintains two gsets – an add-set A and a remove-set R – each holding events $\tau \times \sigma$. It instantiates SetOp at Event $\tau\ \sigma$, so each operation carries an element and a timestamp. map_op routes add and remove to the respective gset via disjoint_product (Section 5.2):

```
def lww_element_set (σ : Type) [PartialOrder τ] [LinearOrder (Event τ σ)] :
   CRDT (SetOp (Event τ σ)) (GSet (Event τ σ) × GSet (Event τ σ)) (SetInterp σ) :=
 map_interpretation (λ ⟨addSet, removeSet⟩ ↦
   let removes := removeSet.toList
   let live := addSet.toList.filter (λ a ↦ ¬ removes.any (λ r ↦ r.o == a.o ∧ (a.t < r.t)))
   { mem s := live.any (λ a ↦ a.o == s), toList := (live.map (·.o)).dedup })
 (map_op (λ | .add elem => .inl elem | .remove elem => .inr elem) (disjoint_product (gset (Event τ σ)) (gset (Event τ σ))))
```

Interpretation resolves each element independently: e is present if an add entry $(e, t) \in A$ has no later remove entry $(e, t') \in R$, so its latest timestamped write was an add. Enumeration goes through toList on both sets; the LinearOrder is required on Event $\tau\ \sigma$ rather than on τ, keeping the order independent of causality. Because τ carries only a *partial* order, two timestamps may be incomparable. When an add-entry (e, t) and a remove-entry (e, t') are concurrent – i.e. neither $t < t'$ nor $t' < t$ holds – the filter retains the add-entry, so e remains in the set, resolving concurrent add and remove in favor of add.

6.1.5 PN-Set. The PN-Set relaxes the 2P-Set's permanence restriction by associating an integer counter with each element rather than a boolean tombstone flag. A strictly positive counter indicates set membership; a negative counter indicates that the element has been removed more times than it has been added.

The PN-Set reuses SetOp as its operation type. map_op maps add to an increment of +1 and remove to a decrement of −1, routed to the appropriate counter sub-CRDT via associate:

```
def pnset (σ : Type) : CRDT (SetOp σ) (Associate σ Int) (SetInterp σ) :=
 map_interpretation (λ s ↦ {
   mem := (· > (0 : Int)) ∘ s.map,
   toList := s.toList.filterMap λ ⟨k, v⟩ ↦ if v > 0 then .some k else .none })
  (map_op (λ | .add elem => ⟨elem, 1⟩ | .remove elem => ⟨elem, −1⟩) (associate σ counter))
```

The state consists of a per-element counter maintained by associate. The interpretation retains only those elements whose counter is strictly positive, restoring the expected set abstraction over the integer-valued internal representation.

6.1.6 Observed-Removed-Set (OR-Set). Both the 2P-Set and the LWW-Element-Set impose restrictions on how elements may be re-added: the 2P-Set forbids re-insertion of tombstoned elements entirely, and the LWW-Element-Set resolves concurrent add/remove by timestamp. The Observed-Removed-Set lifts both restrictions by grounding removal in causality rather than ordering.

The key idea is that each add operation tags the element with a unique token t, producing a pair (e, t) that is inserted into the set. A remove operation does not name a bare element; instead it names the exact set of tagged pairs $\{(e, t_1), \ldots, (e, t_n)\}$ seen by the issuing replica. Hence, only add operations *causally preceding* the remove are cancelled: any concurrent add on another replica will have a fresh token not present in the tombstone, and therefore survives the remove.

Because each remove carries a Finset of tagged pairs rather than a single element, the operation type differs from SetOp:

```
inductive OROp (σ tok : Type) where | add (elem : σ) (token : tok) | remove (observed : Finset (σ × tok))
```

Structurally the OR-Set is a 2P-Set on the tagged type $\sigma \times \text{tok}$, with map_op routing each operation and map_interpretation projecting the surviving tagged pairs back to bare elements:

```
def orset (σ tok : Type) : CRDT (OROp σ tok) (Finset (σ × tok) × Finset (σ × tok)) (Finset σ) :=
 map_interpretation (λ ⟨addSet, tombstone⟩ ↦ (addSet \ tombstone).image Prod.fst)
  (map_op (λ | .add elem token => .inl {(elem, token)} | .remove observed => .inr observed)
   (disjoint_product (union_set (σ × tok)) (union_set (σ × tok))))
```

An element e is visible if a tagged pair (e, t) remains after subtracting tombstones; Finset.image Prod.fst projects surviving pairs to bare values. Concurrent adds from different replicas produce distinct tokens and are therefore unaffected by removes that did not observe them, making the OR-Set the most permissive of the set variants presented here.

6.1.7 Add-Remove Partial Order (List). The list CRDT is the running example developed throughout the paper, and implements a collaboratively editable sequence based on the *Add-Remove Partial Order* of Shapiro et al. [40]. Elements are identified by unique elements of type σ and the sequence order is maintained as a partial order on those elements, encoded as a directed graph of edges between consecutive nodes.

The operation type is a tagged union of Insert and Remove. The state consists of three components: vertices, edges, and a tombstone set of removed vertices. These are maintained by three sub-CRDTs composed via disjoint_product (Section 5.2): insertions jointly update the vertex set and the edge relation, while deletions add the element to the tombstone set via a separate gset CRDT:

```
structure Insert (σ : Type) where
  prev : σ  -- predecessor token
  elem : σ  -- new element token
  next : σ  -- successor token
structure Remove (σ : Type) where
  elem : σ  -- token to remove
```

```
def polist (σ : Type) [LinearOrder σ] [Bot σ] : CRDT
    (Insert σ ⊕ Remove σ) ((GSet σ × Associate σ (GSet σ)) × (GSet σ)) (List σ) :=
  map_interpretation po_traversal
    (disjoint_product
      (map_op (λ ⟨p, v, n⟩ ↦ ⟨v, ⟨p, v, n⟩⟩) (joint_product (gset σ) (po_edges σ)))
      (map_op (λ ⟨o⟩ ↦ o) (gset σ)))
```

Interpretation applies po_traversal, a depth-first traversal from least element $\bot \in \sigma$ that emits a node only once all its predecessors have been visited, yielding live vertices in a total order consistent with the edges' partial order. Concurrent inserts at the same position produce incomparable edges; the LinearOrder on σ provides a deterministic tie-breaker so that replicas produce identical lists after convergence.

6.2 Case Study: Interactive TicTacToe Game

We use our library to represent an entire collaborative game as conflict-free, eventually consistent state. Two players play tic-tac-toe (noughts and crosses) and can fork the state to explore alternative lines of play; a tree retains all forks, and each game node includes a text chat.

The Game Model. Each player is X or O, each cell is empty or occupied, and a nine-cell vector represents the 3×3 board.

```
inductive Player where | X | O
inductive Cell where | Empty | Occupied (p : Player)
def Board := Vector Cell 9
```

Semantic Operation Type. Following the pattern established in Section 5.1, we define an operation type that mirrors the actions a player takes, rather than the raw tagged-union operations of the combinators. Message positions in the chat are identified by Cursor, a totally ordered type with a designated bottom element $\bot$ and top element $\top$ used as sentinels for the head and tail of the list. The place operation updates the board at a path, fork copies it to a new tree path, and chat appends a message under parent in the chat at path.

```
abbrev Cursor
  := WithTop (WithBot Nat)
```

```
inductive TicTacToeOp where
  | place (path : List String) (board : Board)
  | fork  (toPath : List String) (board : Board)
  | chat  (path : List String) (parent : Cursor) (msgId : Cursor) (msg : String)
```

Threaded Chat via the List with Values CRDT. Each game node's chat uses the *list with values* CRDT, which extends the list (Section 6.1.7) by associating an independent value CRDT c with every position:

```
structure MutateList (χ ω : Type) where (cursor : χ) (op : ω)
def polist_with_values (χ : Type) (c : CRDT ω σ γ) :
  CRDT (MutateList χ ω ⊕ Insert χ ⊕ Remove χ)
    ((Associate χ σ) × (Associate χ Int) × (Associate χ (GSet χ))) (List (χ × γ))
```

The operation type extends the list's Insert/Remove with a MutateList case that routes a value operation to the CRDT at a given position. The interpretation zips the traversal order from polist with the interpreted value at each position, yielding a causally ordered List $(\chi \times \gamma)$.

For chat, c is mv_lww_t String, indexed by Cursor. Each cursor identifies a message whose content resides in the LWW register. A chat operation becomes an Insert placing msgId after parent, followed by MutateList writing its text. Because the partial order on cursors in the chat operation encodes the reply-to relationship, replies are always sequenced after the messages they reference, even when delivered out of order across replicas.

Using the Combinators. An associate$_t$ (Line 8) maps List String paths (Line 9) to game nodes, like file-system directories in the fork tree. Each node pairs an mv_lww$_t$ Board – retaining all concurrently updated board states – with a list-with-values chat log via disjoint_product$_t$ (Line 10). Because .chat fans out into two primitive operations, the translation is expressed with traverse$_t$ rather than map_op$_t$:

```
1  def tictactoe_crdt_t [PartialOrder τ] :=
2    traverse_t (λ
3      | TicTacToeOp.place path board              => [⟨path,    .inl board⟩]
4      | TicTacToeOp.fork  toPath board            => [⟨toPath, .inl board⟩]
5      | TicTacToeOp.chat  path parent msgId msg => [
6          ⟨path, .inr $ .inr $ .inl ⟨parent, msgId, ⊤⟩⟩,  -- Insert msgId after parent
7          ⟨path, .inr $ .inl        ⟨msgId, msg⟩⟩ ])     -- MutateList: write msg at msgId
8    (associate_t
9      (List String)
10     (disjoint_product_t (mv_lww_t Board) (polist_with_values_t Cursor (mv_lww_t String))))
```

The composed CRDT's operation, state, and interpretation types (ω, σ, γ) are inferred automatically. .place and .fork each produce a single operation routed to the mv_lww$_t$, while .chat produces two operations routed to the list-with-values CRDT: an Insert to register the new cursor in the partial order, and a MutateList to store the message text via the mv_lww$_t$ String register at that cursor.

Conflict Resolution. Consider a player who is logged into two clients simultaneously and places a piece in different cells at the same game node. Figure 12 illustrates the subsequent conflict resolution process.

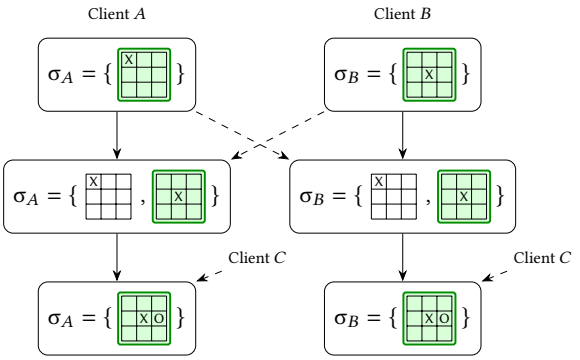


Fig. 12. Two concurrent place operations by the same player.

The two place operations are concurrent, i.e., they carry incomparable timestamps, so neither supersedes the other: the mv_lww$_t$ retains *both* candidate boards in its state.

Once the operations are processed by the respective other client, the conflict is resolved at interpretation: the observed value γ is obtained by selecting the greatest candidate under a total order on Board (derived lexicographically over the cells). This choice is arbitrary but deterministic, which is sufficient for convergence. From the perspective of the client whose board loses, the effect is a *rollback*: the piece they placed disappears once the concurrent operation arrives and both replicas settle on the same winner. Causally ordered writes are unaffected; the total order on states is consulted only to break ties among concurrent events. Notably, the losing move is not destroyed as long as it is not dominated by a new substate, as specified by the multi-value register (Section 3).

6.3 Case Study: Compositional Automerge

We demonstrate that our combinator library is expressive enough to encode a tree-structured CRDT akin to Automerge [3], which supports arbitrary nesting of key–value maps and ordered lists – the structure underlying data formats such as JSON.

Auto Tagged Union. We first introduce auto_tagged_union$_t$, a wrapper around tagged_union$_t$ (Section 5.4) that automatically co-issues a Choice tag with every value operation, eliminating the need for a separate branch-switching operation. Each operation is fanned out by traverse$_t$ into a value sub-operation and a co-issued Choice tag, so the interpretation always knows which branch is active without any extra action from the caller.

```
def auto_tagged_union_t [PartialOrder τ] (c₁ : CRDT_t τ ω₁ σ₁ γ₁) (c₂ : CRDT_t τ ω₂ σ₂ γ₂) :
  CRDT_t τ (ω₁ ⊕ ω₂) ((σ₁ × σ₂) × (Finset (Event τ Choice))) (γ₁ ⊕ γ₂) :=
 traverse_t (λ | l@(.inl _) => [.inl l, .inr Choice.left] | r@(.inr _) => [.inl r, .inr Choice.right]) (tagged_union_t c₁ c₂)
```

Identifiers. An object identifier Oid identifies each tree node; Cursor, the totally ordered sentinel type from the TicTacToe chat (Section 6.2), tracks list positions.

```
abbrev Oid := WithBot Nat -- ⊥ = root node
abbrev Cursor := WithTop (WithBot Nat)
abbrev Key := String
```

Scalars. We assemble scalar and element types from these primitives.

```
def scalar_t := auto_tagged_union_t counter_t ... (mv_lww_t String) ... (mv_lww_t Int) ... (mv_lww_t Bool)
def elem_t := map_op_t (λ | .oid op => .inl op | .scalar op => ... ) (auto_tagged_union_t (mv_lww_t Oid) scalar_t)
```

Lists and Maps. An Automerge list is a polist_with_values$_t$ (Section 6.2) instantiated with elem$_t$ at each cursor position. For better performance, we use an implementation based on pnset$_t$ (Section 6.1.5) instead of gset$_t$ (Section 6.1), which avoids the allocation of the auxiliary tombstone set. An Automerge map pairs an associate$_t$ from Key to elem$_t$ with a pnset$_t$ tracking live keys. In both cases map_op$_t$ exposes a semantic operation type, collapsing the raw tagged-union constructors into readable inductives:

```
def automerge_list_t := map_op_t (λ
  | .update op => .inl op
  | .insert op => .inr $ .inl op
  | .remove op => .inr $ .inr op)
 (polist_with_values_t Cursor elem_t)
```

```
def automerge_map_t := map_op_t (λ
  | .update key op => .inl ⟨key, op⟩
  | .put key => .inr $ .add key
  | .remove key => .inr $ .remove key)
 (disjoint_product_t (associate_t Key elem_t) (pnset_t Key))
```

The Tree. A tree node is a map or list, composed with tagged_union$_t$. An outer associate$_t$ from Oid to nodes forms the tree, allowing updates by Oid without traversal.

The top-level operation type TreeUpdate provides an interface akin to Automerge's public API [3]. Because several operations fan out across multiple sub-CRDTs (e.g. put_map must mark the key as present, store the child Oid, and initialize the child node as a map), the translation uses traverse$_t$:

```
def tree_t := traverse_t (λ | .put_scalar oid key o => [⟨oid, .inl $ .inl $ .put key⟩, ⟨oid, .inl $ .inl $ .update key $ .scalar o⟩] )
 (associate_t Oid (tagged_union_t automerge_map_t automerge_list_t))
```

We elide the remaining cases for brevity; the full definition follows the same pattern. Operations that create a new child node (put_map, put_list, insert_map, insert_list) produce three sub-operations: registering the key or cursor in the parent, storing the child Oid, and tagging the child node as a map or list via Choice. Operations that modify existing data (update_at, remove_key) produce a single sub-operation, and scalar insertions produce two. The use of traverse$_t$ rather than map_op$_t$ accommodates this variable fanout.

Setup. All benchmarks are driven by pre-generated operation files. For each workload, we replay exactly the same file for our Lean implementation and each of the five Automerge releases, applying one operation per transaction. This keeps the logical operation sequence and workload generation identical across implementations while preserving each implementation's own state representation and runtime. The Automerge releases are 0.5.12, 0.6.1, 0.7.4, 0.8.0, and 0.9.0, the latest

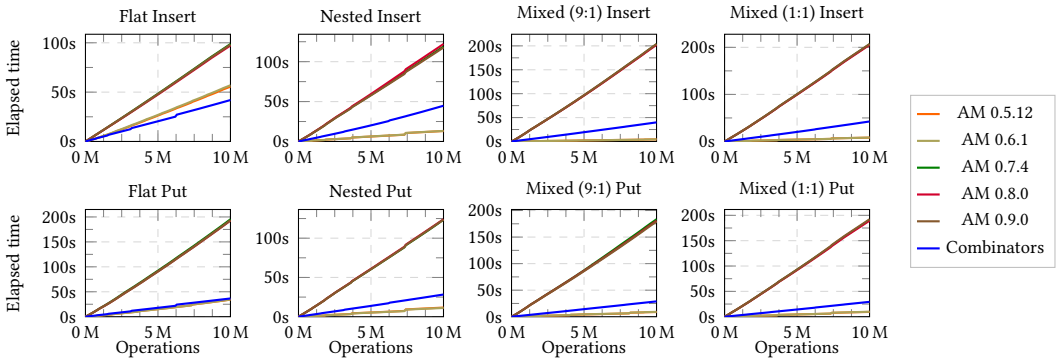


Fig. 13. Synthetic Runtime Benchmarks

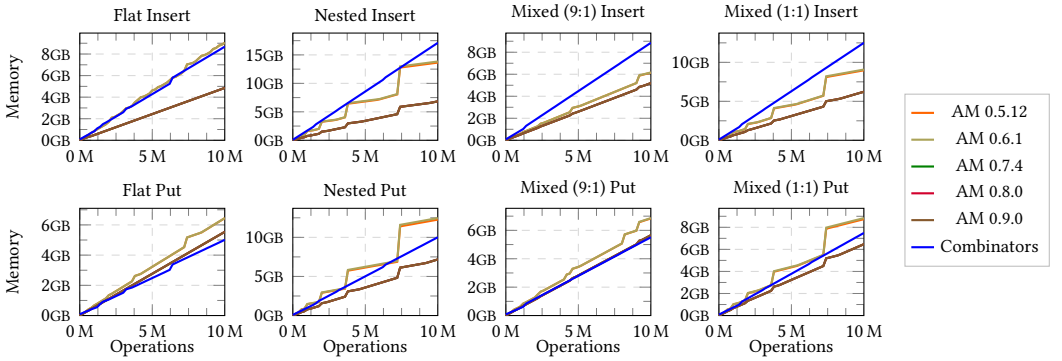


Fig. 14. Synthetic Memory Benchmarks

patches of the represented minor versions. Insert workloads target a list, Put workloads a map. Flat applies every operation to a single top-level object, Nested creates a new nested object for every operation, Mixed interleaves the two at 9:1 and 1:1 ratios, since a purely flat or purely nested stream is an unrealistic extreme; alternating flat and nested operations more closely approximates a heterogeneous workload, with 9:1 modeling mostly-flat usage with occasional nesting and 1:1 modeling a balanced mix. Each reported workload runs up to 10 million operations. We report end-to-end wall-clock time and memory as maximum resident set size (RSS), measured on an AMD EPYC 9654 with swap disabled.

Results. Across these workloads, all implementations scale linearly in the number of operations: per-operation cost stays constant across the full 10 M range (Figure 13). The Automerge releases fall into two performance classes: the older releases (0.5.12, 0.6.1) and the newer ones (0.7.4, 0.8.0, 0.9.0). The latter are about an order of magnitude slower on Put and Mixed workloads. Automerge became more memory-efficient but slower between versions 0.6.1 and 0.7.4, consistent with its announced compression-based memory reduction [4].

Our implementation lies between these classes, outperforming the newer releases in terms of runtime performance on every workload (~2.4–6×) and trailing the older ones on all workloads except Flat Insert. Memory consumption (Figure 14) is likewise linear but differently shaped: our footprint grows smoothly with the live state, whereas Automerge grows in steps, consistent with amortized reallocation of internal buffers. On Insert workloads our memory consumption is largest,

peaking at ~2.5× Automerge 0.9.0 on Nested Insert (17.1 GB vs 6.8 GB); on Put workloads it is comparable, and lowest on Flat Put.

Runtime. Our implementation's linear runtime is not obvious: every effect has pure type $\omega \to \sigma \to \sigma$, suggesting that each of 10 million operations copies state. Lean avoids these copies through a runtime reference-count check [45]: a uniquely referenced consumed value is updated destructively. In the benchmark loop, each effect consumes the sole reference to the previous state and yields the next. The composed pipeline therefore mutates underlying hash maps in place at the cost of an imperative update.

7 Related Work

We review operation- and state-based CRDT construction and verification, programming models that maintain invariants over replicated data, and compositional approaches to CRDTs.

7.1 Conflict-Free Replicated Data Types

Local-first systems [22] give each process full ownership of its data and synchronize when connected. Conflict-free Replicated Data Types (CRDTs) [41] support this model by guaranteeing eventual replica convergence. They have two mutually definable styles: operation-based and state-based; the latter tolerates duplicate messages and embeds causal information [11]. Delta-state CRDTs reduce bandwidth in state-based replication by sending incremental changes [14].

CRDT construction and verification. For a state-based replicated object, Nair et al. [35] verify that weak consistency preserves its application-specific invariants. Their verification is per object; our framework guarantees convergence for every well-typed combinator expression. VeriFx [13] automatically validates functional state-based, delta-state, and operation-based CRDT implementations, then transpiles them to a host language. Composition has no special status: each composed type is verified wholesale. Propel [47] uses a type system to verify algebraic properties of state-based CRDT merges; subsequent work extends it to algebraic laws across domains [48]. These tools target the local obligations assumed by our primitives, while our combinators preserve convergence under composition. Nieto et al. [36] provide Coq [44] libraries for implementing and verifying operation-based CRDTs in separation logic, with manual proofs for each data type. Our combinators confine this effort to the framework: once a primitive commutes, composition adds no obligations.

Quelea [42] lets developers specify "consistency contracts" for applications over eventually consistent stores, based on which an SMT solver maps each operation to the weakest satisfying store-level guarantee (e.g., in Apache Cassandra). It is operation-centric and represents replica state as a POlog of observed effects; our machine-checked interface instead provides orthogonal combinators that do not need to construct a POlog and guarantee convergence by construction.

Katara [26] synthesizes verified state-based CRDTs from sequential reference implementations, guaranteeing convergence by combining simpler building blocks. It restricts synthesis to semilattices built from components such as integers with maximum, Booleans with conjunction, products with lexicographic or pointwise orderings, and maps. In Katara, these combinators define a *search space*: the synthesizer chooses a matching lattice composition and conflict semantics, then verifies the candidate. In our *programming interface*, developers directly choose the composition and conflict semantics (e.g., mv_reg versus lww), obtaining convergence by construction.

Application-level safety and consistency. Mixed-consistency systems combine weak and strong guarantees: CARDs [29] and Carol [30] reason about operations but are not operation-based CRDT constructions; they use guards and refinement types to identify operations requiring coordination. OAC [49] and CTRD [50] instead add strong operations and consistency levels to state-based

replicated data. LoRe [19] and ConLoc [25] enforce application invariants over state-based CRDTs, coordinating when needed; CORDA [17] similarly uses state-based ARDTs, which PRDTs [18] extend to verified consensus protocols. ConSysT [24] instead provides mixed consistency through middleware-replicated objects, following neither CRDT style. These systems target application or protocol safety; our operation-based framework constructs the underlying convergent types.

7.2 CRDT Composition

Prior work explored some of the combinators that we propose individually; we identify the five combinators we present as a unified basis for CRDT composition.

Burckhardt et al. [11] use event relations to specify replicated data types and verify implementations, combining global specification reasoning with implementation-specific reasoning for modularity. Liang and Feng [31] use Abstract Convergence Consistency (ACC) to specify operation-based CRDT operations abstractly, separating verification of implementations and client programs. Nieto et al. [36] build on ACC to prove functional correctness in separation logic, separating application logic from CRDT proofs involving network operations, concurrency control, and mutable state. We build on their product and map combinators. At a denotational level, Leijnse et al. [28] define map, operation-map, and value-map patterns corresponding to our Associate, MapOperation, and MapInterpretation; in contrast, our approach operates on executable effect/state structures.

Bauwens and Gonzalez Boix [8] add routing and recursive resets for dynamically nested pure operation-based CRDTs, implement maps and lists in Flec [7], and verify their framework and representative maps in VeriFx [13]. Their POlog-based framework [6] treats nesting as a dedicated construct; ours derives nested and non-hierarchical designs from convergence-preserving combinators.

Close in spirit, Weidner et al. [46] compose two operation-based CRDTs over shared state using an action, subject to algebraic laws, that reconciles otherwise noncommuting operations, an interaction our products do not directly support. Our five orthogonal combinators instead compose operation, state, and interpretation structure, making the approaches complementary.

ARDTs [23] derive associative, commutative, and idempotent merges for compound lattice states, offering complementary state-based composition. Our operation-based framework composes the operation, state, and interpretation components; with exactly-once application, ac_fun (Section 3) requires associativity and commutativity, but not idempotence.

8 Discussion

We do not claim our five principal combinators are complete in a formal sense; we argue they form a *convergence-preserving* (Theorem 4.1) and *non-redundant* basis for a broad class of operation-based CRDTs that are useful in practice. The evidence has three parts: the combinators are sufficient for the common CRDTs and the more complex case studies evaluated in this work, they subsume the combinators proposed in prior work, and none of them is redundant.

Sufficiency. The evaluation provides evidence across three tiers. Table 1 shows that the standard catalog of Shapiro et al. [40] is reconstructible in 7–13 lines each, with no new convergence proofs: the only side condition any composition carries is map_state's left inverse, which Lean's default tactic discharges automatically. The case study of Section 6.2 shows the motivating case: an application-specific CRDT that appears in no catalog and would otherwise demand a bespoke convergence proof. Finally, the JSON document CRDT in Section 6.3 shows that the basis scales to a structure matching Automerge's document model.

Subsumption. The closest operation-based account is Nieto et al.'s [36] OpLib, which verifies operation-based CRDTs in Aneris separation logic. Its twelve case studies include two combinators:

a product pairing CRDTs and a map indexing one by key. These correspond to our Product and Associate (Sections 4.1 and 4.3). They present these two constructions as useful examples within a catalog, whereas we identify a *basis*: Product and Associate sit alongside MapState, Traverse, and MapInterpretation, none of which their two combinators express. Leijnse et al. [28] identify Associate and MapInterpretation denotationally, but without executable constructions or proofs. Further, their compositions are exercised only one level deep (a map over a simple CRDT, yielding single-column tables), while our basis composes to arbitrary depth, up to a full JSON tree (Section 6). Our contribution is one of *scope* rather than mechanism.

Non-Redundancy. Within our combinator language, no principal combinator is expressible through the other four. Each has a property that no composition of the remaining four can produce: Product is the only combinator that combines the states of *two* CRDTs; MapState the only one that reaches a state type not assembled structurally from the leaves' states by $\times$ and Associate, and the only one that can relayout an existing state at all, since $f' \circ f = id$ makes f an embedding; MapInterpretation the only one that changes the observed value γ while leaving the operation and state fixed; Associate the only one that introduces a dynamic index, turning one CRDT into a keyed family; and Traverse the only one that varies how many operations one external operation induces per component. Table 1 shows empirically that every principal combinator is exercised by at least one evaluated construction. MapState, for instance, is what TaggedUnion3 uses to relayout its state.

Table 1. Combinator dependencies (✓ direct, ○ transitive); citations mark directly corresponding prior constructions.

	Combinators									
	Principal					Derived				
Derived Combinators	Product [36]	Map State	Associate [28, 36]	Traverse	Map Interp. [28]	Map Operation [28]	Disjoint Product	Joint Product	Tagged Union	**LOC**
Map Operation				✓						1
Disjoint Product	✓			✓						1
Joint Product	✓			✓						1
Tagged Union	○			○	✓		✓			5
Tagged Union 3[a]	○	✓		○	○		○		✓	5
Common CRDTs										
Two-Phase-Set	○			○	✓	✓	✓			7
LWW-Element-Set	○			○	✓	✓	✓			10
PN-Set			✓	○	✓	✓				7
OR-Set	○			○	✓	✓	✓			7
Add-Remove Partial Order	○			○	✓	✓	✓	✓		13
Case Studies										
TicTacToe	○			✓	✓	✓	✓			54
Tree	○			✓	✓	✓	✓	✓	✓	150

[a] With Composite State (Section 5.4)

9 Conclusion

CRDTs let replicas converge without coordination and now underpin many collaborative, offline-capable systems. Real applications, however, rarely rely on a single CRDT; they compose sets, maps, counters and trees, and such composition can quietly break convergence guarantees, thrusting developers back into manual proofs.

We present a compositional framework built from five verified combinators – Product, MapState, Associate, Traverse, and MapInterpretation. Because each combinator is proven to preserve commutativity, any nesting of them yields a convergent CRDT *by construction*. All proofs are formalized directly on the implementation in Lean. Case studies show that developers can assemble sophisticated replicas without manual convergence proofs for each composition.

Data-Availability Statement

The artifact accompanying this paper is publicly available on Zenodo [43]. It contains the full Lean 4 combinator library, all CRDT constructions and proofs described in the paper, and the benchmark harness used to produce the Automerge performance comparison in Figures 13 and 14. The library includes machine-checked commutativity proofs for all principal and derived combinators (Sections 4 and 5) and the CRDT primitives presented in Section 3.

Acknowledgments

This work has been co-funded by the Swiss National Science Foundation (SNSF, Grant No. 10001777), by ArmaSuisse Science and Technology, and by the European Union's Horizon research and innovation program (CAPE Project, Grant No. 101189899).

References

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

[2] AntidoteDB. 2021. Overview. https://antidotedb.gitbook.io/documentation/

[3] Automerge Contributors. 2023. Automerge: A JSON-like data structure (a CRDT) for building collaborative applications. https://github.com/automerge/automerge

[4] Automerge Contributors. 2025. Automerge 3.0. https://automerge.org/blog/automerge-3/

[5] Carlos Baquero, Paulo Sérgio Almeida, and Ali Shoker. 2014. Making Operation-Based CRDTs Operation-Based. In *Distributed Applications and Interoperable Systems – 14th IFIP WG 6.1 International Conference, DAIS 2014 (Lecture Notes in Computer Science, Vol. 8460)*, Kostas Magoutis and Peter Pietzuch (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 126–140. doi:10.1007/978-3-662-43352-2_11

[6] Carlos Baquero, Paulo Sérgio Almeida, and Ali Shoker. 2017. Pure Operation-Based Replicated Data Types. arXiv:1710.04469 [cs.DC] doi:10.48550/arXiv.1710.04469

[7] Jim Bauwens and Elisa Gonzalez Boix. 2020. Flec: A Versatile Programming Framework for Eventually Consistent Systems. In *Proceedings of the 7th Workshop on Principles and Practice of Consistency for Distributed Data* (Heraklion, Greece) *(PaPoC '20)*. Association for Computing Machinery, New York, NY, USA, 1–4. doi:10.1145/3380787.3393685

[8] Jim Bauwens and Elisa Gonzalez Boix. 2023. Nested Pure Operation-Based CRDTs. In *37th European Conference on Object-Oriented Programming (ECOOP 2023) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 263)*, Karim Ali and Guido Salvaneschi (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 2:1–2:26. doi:10.4230/LIPIcs.ECOOP.2023.2

[9] Evelyn Borth, Philipp Lersch, and Annette Bieniusa. 2025. Directed Acyclic Graph CRDTs. In *Proceedings of the 12th Workshop on Principles and Practice of Consistency for Distributed Data* (World Trade Center, Rotterdam, Netherlands) *(PaPoC '25)*. Association for Computing Machinery, New York, NY, USA, 30–37. doi:10.1145/3721473.3722141

[10] Sebastian Burckhardt. 2014. Principles of Eventual Consistency. *Found. Trends Program. Lang.* 1, 1–2 (Oct. 2014), 1–150. doi:10.1561/2500000011

[11] Sebastian Burckhardt, Alexey Gotsman, Hongseok Yang, and Marek Zawirski. 2014. Replicated Data Types: Specification, Verification, Optimality. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Diego, CA, USA) *(POPL '14)*. Association for Computing Machinery, New York, NY, USA, 271–284. doi:10.1145/2535838.2535848

[12] Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. 2015. The Lean Theorem Prover (System Description). In *Automated Deduction – CADE-25 – 25th International Conference on Automated Deduction, Berlin, Germany, August 1–7, 2015, Proceedings (Lecture Notes in Computer Science, Vol. 9195)*, Amy P. Felty and Aart Middeldorp (Eds.). Springer International Publishing, Cham, Switzerland, 378–388. doi:10.1007/978-3-319-21401-6_26

[13] Kevin De Porre, Carla Ferreira, and Elisa Gonzalez Boix. 2023. VeriFx: Correct Replicated Data Types for the Masses. In *37th European Conference on Object-Oriented Programming (ECOOP 2023) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 263)*, Karim Ali and Guido Salvaneschi (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 9:1–9:45. doi:10.4230/LIPIcs.ECOOP.2023.9

[14] Vitor Enes, Paulo Sérgio Almeida, Carlos Baquero, and João Leitão. 2019. Efficient Synchronization of State-Based CRDTs. In *Proceedings – 2019 IEEE 35th International Conference on Data Engineering, ICDE 2019* (Macau, China) *(Proceedings – International Conference on Data Engineering)*. IEEE Computer Society, 148–159. doi:10.1109/ICDE.2019.00022

[15] Colin J. Fidge. 1988. Timestamps in message-passing systems that preserve the partial ordering. In *Proceedings of the 11th Australian Computer Science Conference (Australian Computer Science Communications, Vol. 10)*. Canberra, Australia, 56–66.

[16] Alexey Gotsman, Hongseok Yang, Carla Ferreira, Mahsa Najafzadeh, and Marc Shapiro. 2016. 'Cause I'm Strong Enough: Reasoning about Consistency Choices in Distributed Systems. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (St. Petersburg, FL, USA) *(POPL '16)*. Association for Computing Machinery, New York, NY, USA, 371–384. doi:10.1145/2837614.2837625

[17] Julian Haas, Christian Kuessner, Ragnar Mogk, and Mira Mezini. 2025. Think Locally, Act Globally: A Programming Model for Decentralized Applications. *IEEE Internet Computing* 29, 4 (July 2025), 55–64. doi:10.1109/MIC.2025.3618947

[18] Julian Haas, Ragnar Mogk, Annette Bieniusa, and Mira Mezini. 2026. PRDTs: Composable Design and Verification of Consensus Protocols using Replicated Data Types. *Proc. ACM Program. Lang.* OOPSLA (2026), 33 pages. Accepted for publication in the OOPSLA 2026 issue; arXiv preprint. arXiv:2504.05173 [cs.PL] doi:10.48550/arXiv.2504.05173

[19] Julian Haas, Ragnar Mogk, Elena Yanakieva, Annette Bieniusa, and Mira Mezini. 2024. LoRe: A Programming Model for Verifiably Safe Local-First Software. *TOPLAS* 46, 1, Article 2 (Jan. 2024), 26 pages. doi:10.1145/3633769

[20] Jeff Johnson. 2014. How Facebook Scales Big Data Systems. QCon New York 2014. https://www.infoq.com/presentations/scale-facebook-big-data/

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

[22] Martin Kleppmann, Adam Wiggins, Peter van Hardenberg, and Mark McGranaghan. 2019. Local-first software: You own your data, in spite of the cloud. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software* (Athens, Greece) *(Onward! '19)*. Association for Computing Machinery, New York, NY, USA, 154–178. doi:10.1145/3359591.3359737

[23] Christian Kuessner, Ragnar Mogk, Anna-Katharina Wickert, and Mira Mezini. 2023. Algebraic Replicated Data Types: Programming Secure Local-First Software. In *37th European Conference on Object-Oriented Programming (ECOOP 2023) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 263)*, Karim Ali and Guido Salvaneschi (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 14:1–14:33. doi:10.4230/LIPIcs.ECOOP.2023.14

[24] Mirko Köhler, Nafise Eskandani, Pascal Weisenburger, Alessandro Margara, and Guido Salvaneschi. 2020. Rethinking safe consistency in distributed object-oriented programming. *PACMPL* 4, OOPSLA, Article 188 (Nov. 2020), 30 pages. doi:10.1145/3428256

[25] Mirko Köhler, George Zakhour, Pascal Weisenburger, and Guido Salvaneschi. 2025. Consistent Local-First Software: Enforcing Safety and Invariants for Local-First Applications. *IEEE Transactions on Software Engineering* 51, 1 (Jan. 2025), 53–65. doi:10.1109/TSE.2024.3477723

[26] Shadaj Laddad, Conor Power, Mae Milano, Alvin Cheung, and Joseph M. Hellerstein. 2022. Katara: synthesizing CRDTs with verified lifting. *Proc. ACM Program. Lang.* 6, OOPSLA2, Article 173 (Oct. 2022), 29 pages. doi:10.1145/3563336

[27] Leslie Lamport. 1978. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM* 21, 7 (July 1978), 558–565. doi:10.1145/359545.359563

[28] Adriaan Leijnse, Paulo Sérgio Almeida, and Carlos Baquero. 2019. Higher-Order Patterns in Replicated Data Types. In *Proceedings of the 6th Workshop on Principles and Practice of Consistency for Distributed Data* (Dresden, Germany) *(PaPoC '19)*. Association for Computing Machinery, New York, NY, USA, Article 5, 6 pages. doi:10.1145/3301419.3323971

[29] Nicholas V. Lewchenko, Arjun Radhakrishna, Akash Gaonkar, and Pavol Černý. 2018. Conflict-Aware Replicated Data Types. arXiv:1802.08733 [cs.DC] doi:10.48550/arXiv.1802.08733

[30] Nicholas V. Lewchenko, Arjun Radhakrishna, Akash Gaonkar, and Pavol Černý. 2019. Sequential Programming for Replicated Data Stores. *Proceedings of the ACM on Programming Languages* 3, ICFP, Article 106 (Aug. 2019), 28 pages. doi:10.1145/3341710

[31] Hongjin Liang and Xinyu Feng. 2021. Abstraction for conflict-free replicated data types. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI '21)*. Association for Computing Machinery, New York, NY, USA, 636–650. doi:10.1145/3453483.3454067

[32] Dmitry Martyanov. 2018. CRDTs in Production. QCon San Francisco 2018. https://www.infoq.com/presentations/crdt-production/

[33] The Mathlib Community. 2020. The Lean mathematical library. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs* (New Orleans, LA, USA) *(CPP '20)*. Association for Computing Machinery, New York, NY, USA, 367–381. doi:10.1145/3372885.3373824

[34] Friedemann Mattern. 1989. Virtual time and global states of distributed systems. In *Parallel and Distributed Algorithms: Proceedings of the International Workshop on Parallel and Distributed Algorithms (Chateau de Bonas, Gers, France, October 3–6, 1988)*, Michel Cosnard, Yves Robert, Patrice Quinton, and Michel Raynal (Eds.). North-Holland, Amsterdam, The Netherlands, 215–226. https://www.vs.inf.ethz.ch/publ/papers/VirtTimeGlobStates.pdf

[35] Sreeja S. Nair, Gustavo Petri, and Marc Shapiro. 2020. Proving the Safety of Highly-Available Distributed Objects. In *Programming Languages and Systems – 29th European Symposium on Programming, ESOP 2020 (Lecture Notes in Computer Science, Vol. 12075)*, Peter Müller (Ed.). Springer International Publishing, Cham, Switzerland, 544–571. doi:10.1007/978-3-030-44914-8_20

[36] Abel Nieto, Léon Gondelman, Alban Reynaud, Amin Timany, and Lars Birkedal. 2022. Modular verification of op-based CRDTs in separation logic. *Proc. ACM Program. Lang.* 6, OOPSLA2, Article 188 (Oct. 2022), 29 pages. doi:10.1145/3563351

[37] Nuno Preguiça. 2018. Conflict-Free Replicated Data Types: An Overview. arXiv:1806.10254 [cs.DC] doi:10.48550/arXiv.1806.10254

[38] Riak. 2021. Enterprise NoSQL Database. Scalable Database Solutions. Riak. https://riak.com/

[39] Arik Rinberg, Tomer Solomon, Roee Shlomo, Guy Khazma, Gal Lushi, Idit Keidar, and Paula Ta-Shma. 2022. DSON: JSON CRDT using delta-mutations for document stores. *Proc. VLDB Endow.* 15, 5 (Jan. 2022), 1053–1065. doi:10.14778/3510397.3510403

[40] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011. *A Comprehensive Study of Convergent and Commutative Replicated Data Types.* Research Report 7506. INRIA – Centre Paris-Rocquencourt. https://hal.inria.fr/inria-00555588

[41] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011. Conflict-Free Replicated Data Types. In *Stabilization, Safety, and Security of Distributed Systems (Lecture Notes in Computer Science, Vol. 6976)*, Xavier Défago, Franck Petit, and Vincent Villain (Eds.). Springer Berlin Heidelberg, Grenoble, France, 386–400. doi:10.1007/978-3-642-24550-3_29

[42] KC Sivaramakrishnan, Gowtham Kaki, and Suresh Jagannathan. 2015. Declarative Programming over Eventually Consistent Data Stores. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Portland, OR, USA) *(PLDI '15)*. Association for Computing Machinery, New York, NY, USA, 413–424. doi:10.1145/2737924.2737981

[43] Alexander Städing Dominguez, George Zakhour, Pascal Weisenburger, and Guido Salvaneschi. 2026. *Artifact for Composing CRDTs Convergent by Construction.* doi:10.5281/zenodo.21532004

[44] The Coq Development Team. 2024. The Coq Reference Manual – Release 8.19.0. https://coq.inria.fr/doc/V8.19.0/refman

[45] Sebastian Ullrich and Leonardo de Moura. 2021. Counting Immutable Beans: Reference Counting Optimized for Purely Functional Programming. In *Proceedings of the 31st Symposium on Implementation and Application of Functional Languages* (Singapore, Singapore) *(IFL '19)*. Association for Computing Machinery, New York, NY, USA, Article 3, 12 pages. doi:10.1145/3412932.3412935

[46] Matthew Weidner, Heather Miller, and Christopher Meiklejohn. 2020. Composing and Decomposing Op-Based CRDTs with Semidirect Products. *Proc. ACM Program. Lang.* 4, ICFP, Article 94 (Aug. 2020), 27 pages. doi:10.1145/3408976

[47] George Zakhour, Pascal Weisenburger, and Guido Salvaneschi. 2023. Type-Checking CRDT Convergence. *PACMPL* 7, PLDI, Article 162 (June 2023), 24 pages. doi:10.1145/3591276

[48] George Zakhour, Pascal Weisenburger, and Guido Salvaneschi. 2024. Automated Verification of Fundamental Algebraic Laws. *PACMPL* 8, PLDI, Article 178 (June 2024), 24 pages. doi:10.1145/3656408

[49] Xin Zhao and Philipp Haller. 2018. Observable Atomic Consistency for CvRDTs. In *Proceedings of the 8th ACM SIGPLAN International Workshop on Programming Based on Actors, Agents, and Decentralized Control* (Boston, MA, USA) *(AGERE '18)*. Association for Computing Machinery, New York, NY, USA, 23–32. doi:10.1145/3281366.3281372

[50] Xin Zhao and Philipp Haller. 2021. Consistency Types for Replicated Data in a Higher-Order Distributed Programming Language. *The Art, Science, and Engineering of Programming* 5, 2, Article 6 (2021), 30 pages. doi:10.22152/programming-journal.org/2021/5/6

Received 2026-03-24; accepted 2026-08-06