Composition in State-based Replicated Data Types

Carlos Baquero[1], Paulo Sérgio Almeida[1],
Alcino Cunha[1], and Carla Ferreira[2]

[1]HASLab, INESC-TEC & Minho University, Portugal,
{cbm,psa,alcino}@di.uminho.pt
[2]NOVA LINCS, DI, FCT, Universidade NOVA de Lisboa, Portugal,
carla.ferreira@fct.unl.pt

Abstract

Keeping replicated data strongly consistent is convenient when communication is fast and available. In internet-scale distributed systems the reality of high communication latencies and likelihood of partitions, leads developers to adopt more relaxed consistency models, such as eventual consistency. Conflict-free Replicated Data Types, bring structure to the design of eventually consistent data management solutions, by precisely describing the behaviour under concurrent updates and guarantying a path to reconciliation. This paper offers a survey of the mathematical structures that support state based multi-master replication with reconciliation, and shows how state structures and state transformations can be composed to provide data types that are now used in practice in many geo-replicated systems.

1 Introduction

Eventual consistency [25] is a relaxed, and highly available, data consistency model that is often the option of choice in internet-scale distributed systems. The common reasoning is that availability must be maintained, despite outages and partitioning, whereas delayed consistency is acceptable. Replicated data can be independently updated by multiple masters, allowing replicas to temporarily diverge [15], provided that they can eventually be reconciled into a common state. The notion of attaining eventual consistency when updates stop can be traced to R. H. Thomas in [23] *"By mutual consistency we mean that all copies converge to the same state and would be identical should update activity cease"*.

Reconciliation of divergent data has been studied for many years, with strong roots in databases [22, 23] and distributed file-systems [17, 18], often motivated by support of disconnected operation. However, reconciliation algorithms used to be ad-hoc and had to be devised by application layer programmers, typically lacking a sound basis that ensured their correctness and convergence properties. Alternatively, reconciliation can be left to the user, as in version control systems, thus any outcome is possible and reconciliation becomes non deterministic.

Conflict-free Replicated Data Types (CRDTs) [21] are motivated by modern demands from internet-scale systems [3, 14, 16, 24]. Within those systems, they currently serve millions of users world-wide, and bring a more grounded approach to the design of efficient and deterministic reconciliation solutions. They preserve the sequential semantics of the modeled data types, and present a choice among deterministic options when addressing concurrent changes. For instance, when facing concurrent insertion and removal of the same element in a set, different set concurrency semantics can lead to Add-wins or Remove-wins CRDT Sets. While the choice of best CRDT data type implementation is still left to the application designer, each data type is still assured to be correct with respect to its sequential and specific concurrent semantics.

CRDTs support two complementary designs: operation-based and state-based. Operation-based CRDTs require a middleware that provides reliable causal delivery to a known group of replicas, while state-based CRDTs usually only require access to globally unique identifiers and eschew membership information. Due to their additional flexibility, state-based CRDTs have a larger ratio of adoption in industry, and will be the focus of this study.

State-based CRDTs are rooted in the mathematical structure known as join-semilattices (which in this document we will abbreviate to simply lattices). These order structures ensure that: 1) the replicated states of the data types evolve and increase in a partial order in a precise way, as operations are applied, so that the new version subsumes the previous one; 2) all concurrent evolutions can be merged deterministically by the lattice join. In order to understand the building principles of state-based CRDTs it is necessary to understand the basic building blocks of lattices and how lattices can be composed.

In the following sections we will make a bridge linking classic results from order and lattice theory into state-based CRDT construction techniques. We will show how state evolves within a lattice; present several examples of concrete CRDTs; and when possible link them to concrete use cases. We envision two main readership goals: to provide a compact reference of constructions for the benefit of data type developers, and to possibly entice theoreticians to consider a new subject area for practical application of lattice theory.

2 From Sets to Lattices

In this context the most basic structure to define is a set of distinct values. An example is the set of vowels that can defined by extension as $\mathsf{vowels} \doteq \{\mathtt{a}, \mathtt{e}, \mathtt{i}, \mathtt{o}, \mathtt{u}\}$. Elements in a set have no specific order and they only need to be distinguishable.

A partially ordered set, usually known as poset, is a set equipped with a binary relation $\sqsubseteq$ which is reflexive, transitive and anti-symmetric. Given any elements o, p, q in a poset we have:

- (reflexive) $p \sqsubseteq p$
- (transitive) $o \sqsubseteq p \land p \sqsubseteq q \Rightarrow o \sqsubseteq q$
- (anti-symmetric) $p \sqsubseteq q \land q \sqsubseteq p \Rightarrow p = q$

As an example, we can build a poset over the set of vowels by ordering just two elements $\mathtt{a} \sqsubseteq \mathtt{u}$, while the remaining elements are left unordered. These unordered elements are called concurrent.

- (concurrent) $p \parallel q \iff \neg(p \sqsubseteq q \lor q \sqsubseteq p)$

In one extreme, we can build a poset with a total order on the set of vowels with $\mathtt{a} \sqsubseteq \mathtt{e} \sqsubseteq \mathtt{i} \sqsubseteq \mathtt{o} \sqsubseteq \mathtt{u}$. In this example we ordered all elements and thus created a chain, i.e. a set where for any two elements p, q we have either $p \sqsubseteq q$ or $q \sqsubseteq p$.

In the other extreme, we can leave all elements unordered and define a poset that is an antichain, where any two elements are always concurrent. E.g., for the vowels, defining $\sqsubseteq \doteq \{(\mathtt{a}, \mathtt{a}), (\mathtt{e}, \mathtt{e}), (\mathtt{i}, \mathtt{i}), (\mathtt{o}, \mathtt{o}), (\mathtt{u}, \mathtt{u})\}$.

Throughout the paper we will use simple typing rules to clarify how some structures can be obtained from others (by composition or simply by shedding some properties). For example, every poset is obviously also a set:

$$\frac{A : \mathsf{poset}}{A : \mathsf{set}}$$

Given a poset A and a subset S of A, an upper bound of S is an element of A that is greater than or equal to all elements of S. The *least upper bound*, if it exists, is an upper bound that is less than any other upper bound, and therefore, unique. Going back to the chain defined over the set of vowels ($\mathtt{a} \sqsubseteq \mathtt{e} \sqsubseteq \mathtt{i} \sqsubseteq \mathtt{o} \sqsubseteq \mathtt{u}$), considering the subset $\{\mathtt{a}, \mathtt{i}\}$, elements $\mathtt{i}$, $\mathtt{o}$, $\mathtt{u}$ are all upper bounds of the subset, while $\mathtt{i}$ is the least upper bound.

A given poset A is a lattice if there exists a least upper bound for any pair of elements p and q in A, written $p \sqcup q$, being $\sqcup$ called the *join* operator. By definition, this binary join satisfies the following properties:

- (idempotent) $p \sqcup p = p$
- (commutative) $p \sqcup q = q \sqcup p$
- (associative) $o \sqcup (p \sqcup q) = (o \sqcup p) \sqcup q$

We can generalise it to express the least upper bound of any non-empty finite set S in a lattice A as $\bigsqcup S$. Some properties of least upper bounds are:

- (upper bound) $o \sqsubseteq o \sqcup p$
- (least upper bound) $o \sqsubseteq p \land o \sqsubseteq q \Rightarrow o \sqsubseteq p \sqcup q$

There are posets where the join does not exist for all pairs of different elements; these are not lattices. For instance, an antichain is not a lattice, as the join of any pair of different elements does not exist. Another example is bit strings under prefix ordering (e.g., $01 \sqsubseteq 010$) where concurrent elements, e.g., $010 \parallel 100$, are not joinable.

However, having a set and any idempotent, commutative and associative binary operation, which can be called a join, we have a lattice, with the order induced by the join as $p \sqsubseteq q \iff p \sqcup q = q$.

$$\frac{A : \mathsf{lattice}}{A : \mathsf{poset}}$$

When implementing CRDTs, where all possible states must have a join, this can allow skipping the direct implementation of $\sqsubseteq$ and deriving it from $\sqcup$. However, for performance reasons, it might still be advisable to directly implement $\sqsubseteq$ when appropriate.

As will be presented in Section 4.5, a lattice can be obtained from any set A, by using the powerset $\mathcal{P}(\mathsf{A})$ as the supporting set and choosing the order to be set inclusion, which results in the join being set union. In our running example this would be the lattice defined by $\langle \mathcal{P}(\mathsf{vowels}), \subseteq, \cup \rangle$.

As another general rule, any totally ordered set, i.e., any chain, is a lattice, with the join being the maximum.

$$\frac{A : \mathsf{chain}}{A : \mathsf{lattice}}$$

For natural numbers we have the lattice $\langle \mathbb{N}, \leq_{\mathbb{N}}, \mathsf{max} \rangle$. These simple lattices, and others, can be found as building blocks for the Bloom^L system [7], a language supporting eventual consistency without coordination.

Some lattices have a least element, called the bottom element $\bot$. In these cases the least upper bound for the empty set $\bigsqcup \emptyset$ exists, and it is precisely $\bot$. Some properties are:

- (bottom) $\bot \sqsubseteq o$
- (identity) $\bot \sqcup o = o$

Some examples: the lattice formed by the powerset of a given set A has the empty set as bottom, $\langle \mathcal{P}(A), \subseteq, \cup, \emptyset \rangle$, and natural numbers have 0 as bottom. For those lattices that do not have a bottom, it is always possible to add an extra element as bottom, ordered before all others, obtaining a lattice with bottom. We will address this construction when talking about lattice composition by linear sums in Section 4.3. Trivially, lattices with bottom are lattices.

$$\frac{A : \mathsf{lattice}_\bot}{A : \mathsf{lattice}} \qquad \frac{A : \mathsf{chain}_\bot}{A : \mathsf{chain}}$$

2.1 Primitive Lattices

We now introduce a small set of lattices, that will be later useful to construct more complex structures by composition.

Singleton A single element, $\bot$.

$$\frac{}{\mathbb{1} : \mathsf{chain}_\bot}$$

$$\bot \sqsubseteq \bot \qquad \bot \sqcup \bot = \bot$$

Boolean Two elements $\mathbb{B} = \{\mathsf{False}, \mathsf{True}\}$ in a chain, where join is logical $\vee$.

$$\frac{}{\mathbb{B} : \mathsf{chain}_\bot}$$

$$\mathsf{False} \sqsubseteq \mathsf{True} \qquad x \sqcup y = x \vee y \qquad \bot = \mathsf{False}$$

Naturals Natural numbers with maximum as join. We include the 0, thus $\mathbb{N} = \{0, 1, \ldots\}$.

$$\frac{}{\mathbb{N} : \mathsf{chain}_\bot}$$

$$n \sqsubseteq m = n \leq m \qquad n \sqcup m = \mathsf{max}(n, m) \qquad \bot = 0$$

Integers Integers with maximum as join.

$$\frac{}{\mathbb{Z} : \mathsf{chain}}$$

$$n \sqsubseteq m = n \leq m \qquad n \sqcup m = \mathsf{max}(n, m)$$

3 Inflations make CRDTs

State-based CRDTs can be specified by selecting a given lattice to model the state, and choosing the initial state usually as the lattice $\bot$, if there is one. Query operations evaluate an arbitrary function on the state and return a value. Mutation operations do not return values and can only change the state by *inflations*. An inflation f is an endofunction over A that for any value x in A returns an element greater than or equal to x:

- (inflation) $x \sqsubseteq f(x)$

It should be noticed that an inflation is not the same as a monotonic function, $x \sqsubseteq y \Rightarrow f(x) \sqsubseteq f(y)$. (We have noticed this confusion sometimes.) As an example, the function $f(x) = \frac{x}{2}$ on positive reals is monotonic but not an inflation. Inflations can be further classified as non-strict and strict inflations, where a strict inflation is one such that:

- (strict inflation) $x \sqsubset f(x)$

The rules concerning inflations are thus:

$$\frac{\forall x \in a \cdot x \sqsubseteq f(x)}{f : A \xrightarrow{\sqsubseteq} A}$$

$$\frac{\forall x \in a \cdot x \sqsubset f(x)}{f : A \xrightarrow{\sqsubset} A}$$

$$\frac{f : A \xrightarrow{\sqsubset} A}{f : A \xrightarrow{\sqsubseteq} A}$$

A state that is only updated as a result of inflations over its current value will not be modified if joined with some past state: the new state always subsumes the older one. This has important practical implications: state can be transmitted

at-least-once across replicas, since duplicates have no impact. If an old duplicate arrives at a replica, even out of order with more recent states, joining it with the local state will be harmless (a no-op), as its effect will have already been incorporated, and there is no danger of 'going backwards' and losing more recent information.

3.1 Primitive Inflations

Similarly to the primitive lattices introduced above we can define some primitive inflations.

$$\mathsf{id}(x) = x \qquad \frac{}{\mathsf{id} : A \xrightarrow{\sqsubseteq} A}$$

$$\underline{\mathsf{True}}(x) = \mathsf{True} \qquad \frac{}{\underline{\mathsf{True}} : \mathbb{B} \xrightarrow{\sqsubseteq} \mathbb{B}}$$

$$\mathsf{succ}(x) = x + 1 \qquad \frac{}{\mathsf{succ} : \mathbb{N} \xrightarrow{\sqsubset} \mathbb{N}}$$

3.2 Sequential Composition

Inflations can be composed sequentially. As long as there is at least one strict inflation in the composition, we obtain a strict inflation.

$$(f \bullet g)(x) = f(g(x))$$

$$\frac{f : A \xrightarrow{\sqsubseteq} A \quad g : A \xrightarrow{\sqsubseteq} A}{f \bullet g : A \xrightarrow{\sqsubseteq} A}$$

$$\frac{f : A \xrightarrow{\sqsubseteq} A \quad g : A \xrightarrow{\sqsubset} A}{f \bullet g : A \xrightarrow{\sqsubset} A} \qquad \frac{f : A \xrightarrow{\sqsubset} A \quad g : A \xrightarrow{\sqsubseteq} A}{f \bullet g : A \xrightarrow{\sqsubset} A}$$

4 Lattice Compositions

Since we are interested in creating lattices we consider a few composition techniques that are known to derive lattices. While in some cases they build from other lattices, in others they can derive lattices from simpler structures.

4.1 Product

The product $\times$, or pair construction, derives a lattice formed by pairs of other lattices. It can be applied recursively and derive a composition from a sequence of lattices, where operations are applied in point-wise order.

$$\frac{A : \mathsf{lattice} \qquad B : \mathsf{lattice}}{A \times B : \mathsf{lattice}}$$

$$(x_1, y_1) \sqsubseteq (x_2, y_2) = x_1 \sqsubseteq x_2 \wedge y_1 \sqsubseteq y_2$$

$$(x_1, y_1) \sqcup (x_2, y_2) = (x_1 \sqcup x_2, y_1 \sqcup y_2)$$

The construction also extends to lattices with bottom.

$$\frac{A : \mathsf{lattice}_\bot \qquad B : \mathsf{lattice}_\bot}{A \times B : \mathsf{lattice}_\bot}$$

$$\bot = (\bot, \bot)$$

As an example, the underlying lattice structure of a version vector [15] among three replica nodes is composable by $\mathbb{N} \times \mathbb{N} \times \mathbb{N}$ with $\bot = (0, 0, 0)$.

Bellow are the properties of inflations over products. A strict inflation on one of the components leads to an overall strict inflation.

$$(f \times g)(x, y) = (f(x), g(y))$$

$$\frac{f : A \xrightarrow{\sqsubseteq} A \quad g : B \xrightarrow{\sqsubseteq} B}{f \times g : A \times B \xrightarrow{\sqsubseteq} A \times B}$$

$$\frac{f : A \xrightarrow{\sqsubseteq} A \quad g : B \xrightarrow{\sqsubset} B}{f \times g : A \times B \xrightarrow{\sqsubset} A \times B} \qquad \frac{f : A \xrightarrow{\sqsubset} A \quad g : B \xrightarrow{\sqsubseteq} B}{f \times g : A \times B \xrightarrow{\sqsubset} A \times B}$$

Classic causality based event ordering mechanisms can be described by lattices that evolve by strict inflations. Lamport scalar logical clocks [13] are described by the $\mathbb{N}$ lattice where each event number is generated locally by strict inflation, and received remote clocks are merged-in by join. Similarly, vector clocks share the same structure as version vectors, a product composition of $\mathbb{N}$ lattices, with local events generated by strict inflation of the local entry.

4.2 Lexicographic Product

The ⊠ construct builds a lexicographic order from its source lattices. Components to the left are more significant and, unless they are equal, they filter out further comparisons towards the right side.

$$\frac{A : \mathsf{lattice} \qquad B : \mathsf{lattice}_\bot}{A \boxtimes B : \mathsf{lattice}} \qquad \frac{A : \mathsf{lattice}_\bot \qquad B : \mathsf{lattice}_\bot}{A \boxtimes B : \mathsf{lattice}_\bot}$$

$$(x_1, y_1) \sqsubseteq (x_2, y_2) = x_1 \sqsubseteq x_2 \vee (x_1 = x_2 \wedge y_1 \sqsubseteq y_2)$$

$$(x_1, y_1) \sqcup (x_2, y_2) = \begin{cases} (x_1, y_1) & \textbf{if } x_2 \sqsubset x_1 \\ (x_2, y_2) & \textbf{if } x_1 \sqsubset x_2 \\ (x_1, y_1 \sqcup y_2) & \textbf{if } x_1 = x_2 \\ (x_1 \sqcup x_2, \bot) & \textbf{if } x_1 \parallel x_2 \end{cases}$$

$$\bot = (\bot, \bot)$$

If the left component is a chain, often the case in practical uses based on timestamps or scalar logical clocks, then the right one can be any lattice (without requiring ⊥) as the fourth clause of the join definition never applies.

$$\frac{A : \mathsf{chain} \qquad B : \mathsf{lattice}}{A \boxtimes B : \mathsf{lattice}}$$

And, if the right component is also a chain the composition is a chain.

$$\frac{A : \mathsf{chain} \qquad B : \mathsf{chain}}{A \boxtimes B : \mathsf{chain}}$$

Some properties of inflations on lexicographic products are:

$$(f \boxtimes g)(x, y) = (f(x), g(y))$$

$$\frac{f : A \xrightarrow{\sqsubseteq} A \quad g : B \xrightarrow{\sqsubseteq} B}{f \boxtimes g : A \boxtimes B \xrightarrow{\sqsubseteq} A \boxtimes B}$$

$$\frac{f : A \xrightarrow{\sqsubseteq} A \quad g : B \xrightarrow{\sqsubset} B}{f \boxtimes g : A \boxtimes B \xrightarrow{\sqsubset} A \boxtimes B} \qquad \frac{f : A \xrightarrow{\sqsubset} A \quad g : B \longrightarrow B}{f \boxtimes g : A \boxtimes B \xrightarrow{\sqsubset} A \boxtimes B}$$

Notice that if we apply a strict inflation to the left component, then the right component can be transformed by any function even if non inflationary. In practice this allows resetting the right component after strictly inflating the left; we will see this in Section 5.2 when building lexicographic counters.

The abstraction provided by the lexicographic product is at the core of many practical systems that use *last-writer-wins* approaches to manage concurrent data updates [11]. By using fine grained timestamps in the left side and keeping node clocks as closely synchronized as possible across system nodes, one can expect strict inflations on the left as timestamps increase with time. When merging, higher timestamp values will determine the outcome of the join.

4.3 Linear Sum

The next composition, linear sum $\oplus$, picks two lattices, left and right, and creates a new lattice where any element from the left lattice is always ordered as less than any element in the right lattice. In the resulting set the elements are tagged with a label that identifies from which source lattice they come from. i.e., $\mathsf{Left}\ a$ means that element a comes from the left lattice and is now named $\mathsf{Left}\ a$. Tagging also ensures that the sets supporting each lattice can have elements in common.

$$\frac{A : \mathsf{lattice} \qquad B : \mathsf{lattice}}{A \oplus B : \mathsf{lattice}} \qquad \frac{A : \mathsf{lattice}_\bot \qquad B : \mathsf{lattice}}{A \oplus B : \mathsf{lattice}_\bot}$$

$$\begin{array}{ll} \mathsf{Left}\ x \sqsubseteq \mathsf{Left}\ y = x \sqsubseteq y & \mathsf{Left}\ x \sqcup \mathsf{Left}\ y = \mathsf{Left}\ (x \sqcup y) \\ \mathsf{Right}\ x \sqsubseteq \mathsf{Right}\ y = x \sqsubseteq y & \mathsf{Right}\ x \sqcup \mathsf{Right}\ y = \mathsf{Right}\ (x \sqcup y) \\ \mathsf{Left}\ x \sqsubseteq \mathsf{Right}\ y = \mathsf{True} & \mathsf{Left}\ x \sqcup \mathsf{Right}\ y = \mathsf{Right}\ y \\ \mathsf{Right}\ x \sqsubseteq \mathsf{Left}\ y = \mathsf{False} & \mathsf{Right}\ x \sqcup \mathsf{Left}\ y = \mathsf{Right}\ x \end{array}$$

$$\bot = \mathsf{Left}\ \bot$$

A possible use of this construction is to add a $\bot$ to a lattice that did not had one. For instance $\mathbb{1} \oplus \mathbb{R}$ can add a special element, e.g. nil, that is ordered as less than any real number. The same construction can also be used to add a top element $\top$ to a lattice, that can act as a tombstone that stops lattice evolution.

Some properties of inflations on sums are:

$$\begin{array}{l} (f \oplus g)(\mathsf{Left}\ x) = \mathsf{Left}\ f(x) \\ (f \oplus g)(\mathsf{Right}\ x) = \mathsf{Right}\ g(x) \end{array}$$

$$\frac{f : A \xrightarrow{\sqsubseteq} A \quad g : B \xrightarrow{\sqsubseteq} B}{f \oplus g : A \oplus B \xrightarrow{\sqsubseteq} A \oplus B}$$

$$\frac{f : A \xrightarrow{\sqsubset} A \quad g : B \xrightarrow{\sqsubset} B}{f \oplus g : A \oplus B \xrightarrow{\sqsubset} A \oplus B}$$

4.4 Functions and Maps

The function space $A \to B$ is a lattice, obtained by combining a set A with a lattice B, and using pointwise comparison and join.

$$\frac{A : \mathsf{set} \qquad B : \mathsf{lattice}}{A \to B : \mathsf{lattice}} \qquad \frac{A : \mathsf{set} \qquad B : \mathsf{lattice}_\bot}{A \to B : \mathsf{lattice}_\bot}$$

$$f \sqsubseteq g = \forall x \in A \cdot f(x) \sqsubseteq g(x) \qquad (f \sqcup g)(x) = f(x) \sqcup g(x)$$

$$\bot(x) = \bot$$

Many CRDTs are based on partial functions, i.e., maps $K \hookrightarrow V$, where K is any set of keys, and V is any lattice with bottom. Such maps are sometimes used to (efficiently) represent total functions, assuming that keys which are not present in the map implicitly yield bottom.

$$m(k) = \begin{cases} v & \textbf{if } (k, v) \in m \\ \bot & \textbf{otherwise} \end{cases}$$

This view of maps as functions also allows us to reuse the respective definition for join.

$$\frac{K : \mathsf{set} \qquad V : \mathsf{lattice}_\bot}{K \hookrightarrow V : \mathsf{lattice}_\bot}$$

An example of a map from vowels to integers $\mathsf{vowels} \hookrightarrow \mathbb{N}$ is $m = \{\mathtt{a} \mapsto 3, \mathtt{i} \mapsto 5\}$. Viewing this map as function we could query for $m(\mathtt{u})$ which yields 0.

We now define some inflations over maps. The first applies an inflation to all values in the co-domain and thus inflates the whole map.

$$\mathsf{map}(f)(m) = \{(k, f(v)) \mid (k, v) \in m\}$$

$$\frac{f : V \xrightarrow{\sqsubseteq} V}{\mathsf{map}(f) : (K \hookrightarrow V) \xrightarrow{\sqsubseteq} (K \hookrightarrow V)}$$

The second applies an inflation to the value for a given key.

$$\mathsf{apply}_k(f)(m) = m\{k \mapsto f(m(k))\}$$

Note that if the key is missing the function is applied to $\bot$.

$$\frac{f : V \xrightarrow{\sqsubseteq} V}{\mathsf{apply}_k(f) : (K \hookrightarrow V) \xrightarrow{\sqsubseteq} (K \hookrightarrow V)}$$

Having maps we can refine the modeling of fixed size vector clocks that was based on products of $\mathbb{N}$. A dynamic vector clock can be obtained by mapping node identifiers to $\mathbb{N}$ lattices, as described in [7].

4.5 Sets and Multisets

As we have seen in Section 2, given any $\mathsf{set}\ A$ it is possible to derive a lattice with bottom by using the set of all possible subsets, the powerset $\mathcal{P}(A)$.

The powerset can also be defined by a function that maps each set element to a boolean that states its presence in the subset. This composition is very general since it can produce a lattice with bottom from any set.

$$\mathcal{P}(A) \cong A \to \mathbb{B}$$

Given that $\mathbb{B}$ is a lattice with bottom, from the previous section we know that function $A \to \mathbb{B}$ is also a lattice with bottom. Moreover, the respective order relation and join operator are obtained by construction and are equivalent to the expected.

$$\frac{A : \mathsf{set}}{\mathcal{P}(A) : \mathsf{lattice}_\perp}$$

$$a \sqsubseteq b = a \subseteq b \qquad a \sqcup b = a \cup b \qquad \perp = \{\}$$

A natural extension is to represent *multisets* by mapping the domain set to naturals, instead of booleans.

$$\frac{A : \mathsf{set}}{\mathcal{P}_m(A) : \mathsf{lattice}_\perp}$$

$$\mathcal{P}_m(A) \cong A \to \mathbb{N}$$

Once again, by the properties of lattice composition, we get that function space $A \to \mathbb{N}$ is a lattice with bottom and both the order relation and join operator are provided by construction.

Given that both $\mathbb{B}$ and $\mathbb{N}$ are lattices with bottom, actual CRDTs for sets and multisets use maps to represent functions, as discussed in the previous section. The generic inflations defined for maps can be used here to define an inflation that adds an element e to a given set s.

$$\mathsf{add}(e)(s) = \mathsf{apply}_e(\underline{\mathsf{True}})(s)$$

Likewise, to add an element to a multiset one increments the element count, having a strict inflation.

$$\mathsf{add}(e)(s) = \mathsf{apply}_e(\mathsf{succ})(s)$$

4.6 Maximal Elements

A *down-set* (or order ideal) D of a poset P, is downward closed set, according to $\sqsubseteq$, of elements in P; i.e., if $x \in D$ and $y \sqsubseteq x$, then $y \in D$. Down-sets are useful in many situations, e.g., to represent *causal histories* [19] of all events in the past, up to a given point. Down-sets are also closed under set union, which means that the set of down-sets $\mathcal{D}(A)$ of a poset A is a lattice with bottom (similarly to the powerset for sets), with the usual set inclusion for order and union for join.

$$\frac{A : \mathsf{poset}}{\mathcal{D}(A) : \mathsf{lattice}_\perp}$$

But as they tend to get very large, they are used more as a modelling device, than as an actual construct in implementations. However, a down-set can be more compactly represented by the set of its maximal elements, which is an antichain.

$$\mathsf{maximal}(S) = \{x \in S \mid \nexists y \in S \cdot x \sqsubset y\}$$

This means that, starting from a poset A, we can obtain a lattice with bottom, isomorphic to $\mathcal{D}(A)$, which we call $\mathcal{M}(A)$: the lattice of maximal elements.

$$\mathcal{M}(A) = \{\mathsf{maximal}(S) \mid S \in \mathcal{P}(A)\}$$

$$\frac{A : \mathsf{poset}}{\mathcal{M}(A) : \mathsf{lattice}_\perp}$$

$$\mathcal{M}(A) \cong \mathcal{D}(A)$$

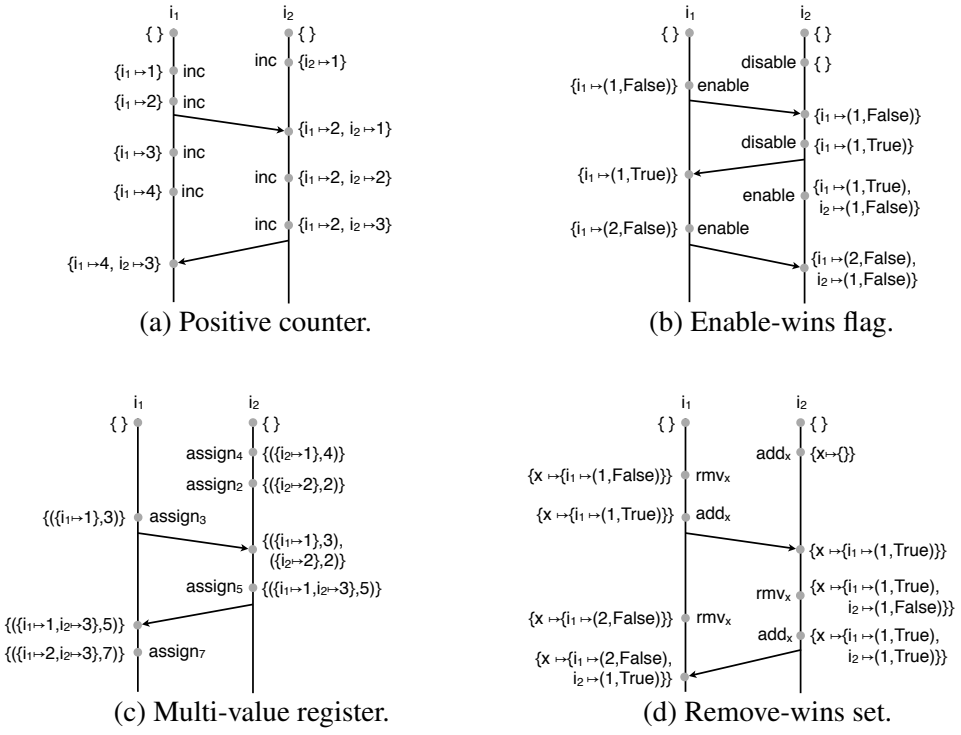
The definitions of join and the order come directly from the isomorphism. Upon a join, given two antichains, all elements that are concurrent are kept, but any element that is subsumed by a greater element is removed.

$$a \sqcup b = \mathsf{maximal}(a \cup b)$$

$$a \sqsubseteq b = \forall x \in a \cdot \exists y \in b \cdot x \sqsubseteq y$$

$$\perp = \{\}$$

In [7] a similar structure, using vector clocks to capture poset ordering, is described as a ldom lattice and referred to as a *dominating set*.



(a) Positive counter. (b) Enable-wins flag.

(c) Multi-value register. (d) Remove-wins set.

Figure 1: Example executions.

The maximal elements construction is the basis for the creation of *multi-value registers* that can store both single values and multiple values, when concurrent assignment occurs (see Section 5.7). This is the core data-type for tracking updates to shopping carts in the original Amazon Dynamo framework [10], and occurs in derived implementations such as the Riak Key-Value Store [4].

5 Abridged Catalog

In order to exemplify the composition constructs we present a small set of example CRDTs. Simple query functions are included and all mutators are inflations. Notice that join does not need to be defined as it follows from the composition rules that were introduced. Figure 1 shows example executions of most CRDTs discussed in this section.

5.1 Positive Counter

This simple form of counter can only increase. Replica nodes must have access to unique ids among a set I; each can only increment its position in a map of

ids to integers. While increment mutators are parametrized by id i the query is anonymous and simply inspects the state.

$$\mathsf{PCounter} = I \hookrightarrow \mathbb{N}$$

$$\begin{aligned} \mathsf{inc}_i(a) &= \mathsf{apply}_i(\mathsf{succ})(a) \\ \mathsf{value}(a) &= \sum\{v \mid (i, v) \in a\} \end{aligned}$$

Notice that if a given node does not yet have an entry in the map and increments, then succ applies over $\perp$, which for $\mathbb{N}$ was defined to be 0.

5.2 Positive and Negative Counter

This variation allows for both increments and decrements. A solution is to pair two positive counters and consider the right side as negative. We use the standard functions fst() and snd() to respectively access the left and right elements of a pair.

$$\mathsf{PNCounter} = (I \hookrightarrow \mathbb{N}) \times (I \hookrightarrow \mathbb{N})$$

$$\begin{aligned} \mathsf{inc}_i(a) &= (\mathsf{apply}_i(\mathsf{succ})(\mathsf{fst}(a)), \mathsf{snd}(a)) \\ \mathsf{dec}_i(a) &= (\mathsf{fst}(a), \mathsf{apply}_i(\mathsf{succ})(\mathsf{snd}(a))) \\ \mathsf{value}(a) &= \sum\{v \mid (i, v) \in \mathsf{fst}(a)\} - \sum\{v \mid (i, v) \in \mathsf{snd}(a)\} \end{aligned}$$

An alternative way to obtain a similar result is to use a lexicographic pair and have the first element incremented when one needs to update the count on the second element.

$$\mathsf{LexCounter} = I \hookrightarrow \mathbb{N} \boxtimes \mathbb{Z}$$

$$\begin{aligned} \mathsf{inc}_i(a) &= \mathsf{apply}_i(\mathsf{id} \boxtimes \mathsf{succ})(a) \\ \mathsf{dec}_i(a) &= \mathsf{apply}_i(\mathsf{succ} \boxtimes \mathsf{pred})(a) \\ \mathsf{value}(a) &= \sum\{\mathsf{snd}(v) \mid (i, v) \in a\} \end{aligned}$$

$$\mathsf{pred}(x) = x - 1$$

While the PNCounter was one of the first CRDTs to be added to a production database, in Riak 1.4 [4], the competing Cassandra database had its own counter implementations based on the LWW strategy. Interestingly it proved to be difficult to avoid semantic anomalies in the behaviour of those early counters, and since Cassandra 2.1, a new counter was introduced [8] in line with the LexCounter.

5.3 Enable-wins Flag

A boolean flag that can be flipped, implemented in Riak under the name flag data type [3]. It uses a lexicographic pair per replica, where the left (more significant) element is a grow-only counter and the right element is a boolean. Enabling the flag increases the counter and resets the flag to False, for the replica entry; disabling the flag sets all booleans to True (maintaining the counters). The fact that only the enable operation increases the counter, ensures that this operation takes precedence over the disable operation. Flag starts disabled.

$$\mathsf{EWFlag} = I \hookrightarrow \mathbb{N} \boxtimes \mathbb{B}$$

$$\begin{aligned} \mathsf{enable}_i(a) &= \mathsf{apply}_i(\mathsf{succ} \boxtimes \underline{\mathsf{False}})(a) \\ \mathsf{disable}(a) &= \mathsf{map}(\mathsf{id} \boxtimes \underline{\mathsf{True}})(a) \end{aligned}$$

$$\mathsf{value}(a) = \exists i, n \cdot (i, (n, \mathsf{False})) \in a$$

5.4 Disable-wins Flag

The disable-wins flag is a dual construction of the enable-wins flag, and uses the same state lattice. Disabling the flag increases the counter, while enabling the flag sets all boolean to True. Flag starts enabled.

$$\mathsf{DWFlag} = I \hookrightarrow \mathbb{N} \boxtimes \mathbb{B}$$

$$\begin{aligned} \mathsf{disable}_i(a) &= \mathsf{apply}_i(\mathsf{succ} \boxtimes \underline{\mathsf{False}})(a) \\ \mathsf{enable}(a) &= \mathsf{map}(\mathsf{id} \boxtimes \underline{\mathsf{True}})(a) \end{aligned}$$

$$\mathsf{value}(a) = \nexists\, i, n \cdot (i, (n, \mathsf{False})) \in a$$

5.5 Add-wins Set

A set with add-wins semantics can be derived by creating unique tokens whenever a new element is inserted, using for that a grow-only counter per replica, and canceling these tokens, by setting a boolean to True, upon removal. Only elements supported by non-canceled tokens are considered to be in the set.

$$\mathsf{AWSet} = E \hookrightarrow I \hookrightarrow \mathbb{N} \boxtimes \mathbb{B}$$

$$
\begin{aligned}
\mathsf{add}_{e,i}(a) &= \mathsf{apply}_e(\mathsf{apply}_i(\mathsf{succ} \boxtimes \underline{\mathsf{False}}))(a) \\
\mathsf{rmv}_e(a) &= \mathsf{apply}_e(\mathsf{map}(\mathsf{id} \boxtimes \underline{\mathsf{True}}))(a)
\end{aligned}
$$

$$
\mathsf{member}_e(a) = \exists (e, m) \in a \cdot \exists i, n \cdot (i, (n, \mathsf{False})) \in m
$$

5.6 Remove-wins Set

A set with remove-wins semantics is derived by a dual construction to the previous one, while sharing the same state lattice. Removal creates unique tokens, and additions need to cancel all remove tokens that are visible in the state.

$$
\mathsf{RWSet} = E \hookrightarrow I \hookrightarrow \mathbb{N} \boxtimes \mathbb{B}
$$

$$
\begin{aligned}
\mathsf{rmv}_{e,i}(a) &= \mathsf{apply}_e(\mathsf{apply}_i(\mathsf{succ} \boxtimes \underline{\mathsf{False}}))(a) \\
\mathsf{add}_e(a) &= \mathsf{apply}_e(\mathsf{map}(\mathsf{id} \boxtimes \underline{\mathsf{True}}))(a)
\end{aligned}
$$

$$
\mathsf{member}_e(a) = \exists (e, m) \in a \cdot \nexists i, n \cdot (i, (n, \mathsf{False})) \in m
$$

5.7 Multi-value Register

A non-optimized multi-value register can be derived by lexicographic coupling a version vector clock $I \hookrightarrow \mathbb{N}$ with a payload value V. When a new value v is to be assigned, a new clock, greater than all visible clocks in the state, is created and coupled with the value. These pairs are kept in an antichain of maximal elements. Thus, upon merge, concurrently assigned values will be collected, but any subsequent assignment will again reduce the state to a single pair.

$$
\mathsf{MVRegister} = \mathcal{M}((I \hookrightarrow \mathbb{N}) \boxtimes V)
$$

$$
\begin{aligned}
\mathsf{assign}_{v,i}(a) &= \{\mathsf{apply}_i(\mathsf{succ})(\bigsqcup \{c \mid (c, v') \in a\}) \boxtimes v\} \\
\mathsf{values}(a) &= \{v \mid (c, v) \in a\}
\end{aligned}
$$

Notice that the value is never updated without creating a new clock. Thus, lexicographic comparison (needed for the operation of the maximals join) is always

decided by the first component, and V can be any opaque payload with no need for a partial order.

It is possible to improve the multi-value register construction, by keeping single tags with each value entry and storing a common causal context [26], and by compacting concurrent assignment of identical values [6]. Making such an improvement, while automatically deriving the join by lattice composition is still an open problem.

6 Closing Remarks

This report collects several composition techniques for lattices, adopts the notion of inflation and shows how it applies to the specification of state based CRDTs over lattices. Most of the lattice compositions are very standard techniques from order theory [9]. An early version of this work was presented at Schloss Dagstuhl under the title *Composition of Lattices and CRDTs* and the summary of the presentation is available at [12]. Most of the CRDT constructions used here are influenced by work in [2, 5–7, 20, 21].

The CRDTs selected for this small abridged catalog illustrate the potential of lattice composition, but do not cover the whole spectrum of known CRDTs, neither aims to be optimized. Further analysis on efficient implementations and optimality results can be found on [1, 6].

Acknowledgements. The work presented was partially supported by FCT-MCTES-PT NOVA LINCS project (UID/CEC/04516/2013), EU FP7 SyncFree project (609551), EU H2020 LightKone project (732505), and SMILES line in project TEC4Growth (NORTE-01-0145-FEDER-000020).

References

[1] Paulo Sérgio Almeida, Ali Shoker, and Carlos Baquero. Delta state replicated data types. *Journal of Parallel and Distributed Computing*, 2017. http://dx.doi.org/10.1016/j.jpdc.2017.08.003.

[2] Carlos Baquero, Paulo Sérgio Almeida, and Ali Shoker. Making operation-based CRDTs operation-based. In *Proceedings of Distributed Applications and Interoperable Systems: 14th IFIP WG 6.1 International Conference, (DAIS'14)*, pages 126–140. Springer, 2014. http://dx.doi.org/10.1007/978-3-662-43352-2_11.

[3] Basho. Riak datatypes, Retrieved 22-Dec-2015. http://github.com/basho.

[4] Basho. Riak 1.4, Retrieved 4-Jan-2016. https://github.com/basho/riak/blob/1.4/RELEASE-NOTES.md.

[5] Annette Bieniusa, Marek Zawirski, Nuno Preguiça, Marc Shapiro, Carlos Baquero, Valter Balegas, and Sérgio Duarte. Brief announcement: Semantics of eventually consistent replicated sets. In *Distributed Computing: 26th International Symposium (DISC'12)*, volume 7611 of *LNCS*, pages 441–442. Springer, 2012. http://dx.doi.org/10.1007/978-3-642-33651-5_48.

[6] Sebastian Burckhardt, Alexey Gotsman, Hongseok Yang, and Marek Zawirski. Replicated data types: Specification, verification, optimality. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'14)*, pages 271–284. ACM, 2014. http://doi.acm.org/10.1145/2535838.2535848.

[7] Neil Conway, William R. Marczak, Peter Alvaro, Joseph M. Hellerstein, and David Maier. Logic and lattices for distributed programming. In *Proceedings of the 3rd ACM Symposium on Cloud Computing (SoCC'12)*, pages 1:1–1:14. ACM, 2012. http://doi.acm.org/10.1145/2391229.2391230.

[8] Datastax. What's New in Cassandra 2.1: Better Implementation of Counters, Retrieved 4-Jan-2016. http://www.datastax.com/dev/blog/whats-new-in-cassandra-2-1-a-better-implementation-of-counters.

[9] Brian A. Davey and Hilary A. Priestley. *Introduction to Lattices and Order (2. ed.)*. Cambridge University Press, 2002.

[10] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. In *Proceedings of 21st ACM SIGOPS Symposium on Operating Systems Principles (SOSP'07)*, pages 205–220. ACM, 2007. http://doi.acm.org/10.1145/1294261.1294281.

[11] Jonathan Ellis. Why cassandra doesn't need vector clocks, Datastax, Retrieved 4-Jul-2017. https://www.datastax.com/dev/blog/why-cassandra-doesnt-need-vector-clocks.

[12] Bettina Kemme, André Schiper, G. Ramalingam, and Marc Shapiro. Dagstuhl seminar review: Consistency in distributed systems. *SIGACT News*, 45(1):67–89, 2014. http://doi.acm.org/10.1145/2596583.2596601.

[13] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978. http://doi.acm.org/10.1145/359545.359563.

[14] Michael Owen. Using Erlang, Riak and the ORSWOT CRDT at bet365 for scalability and performance, Retrieved 17-Jul-2017. http://www.erlang-factory.com/euc2015/michael-owen.

[15] D. S. Parker, G. J. Popek, G. Rudisin, A. Stoughton, B. J. Walker, E. Walton, J. M. Chow, D. Edwards, S. Kiser, and C. Kline. Detection of mutual inconsistency in distributed systems. *IEEE Trans. Softw. Eng.*, 9(3):240–247, 1983. http://dx.doi.org/10.1109/TSE.1983.236733.

[16] Peter Bourgon. Consistency without Consensus: CRDTs in Production at SoundCloud, Retrieved 22-Dec-2015. http://www.infoq.com/presentations/crdt-soundcloud.

[17] Peter Reiher, John Heidemann, David Ratner, Greg Skinner, and Gerald Popek. Resolving file conflicts in the ficus file system. In *Proceedings of the USENIX Summer 1994 Technical Conference on USENIX Summer 1994 Technical Conference (USTC'94)*, pages 12–12. USENIX Association, 1994. http://dl.acm.org/citation.cfm?id=1267257.1267269.

[18] Mahadev Satyanarayanan, James J. Kistler, Puneet Kumar, Maria E. Okasaki, Ellen H. Siegel, and David C. Steere. Coda: A highly available file system for a distributed workstation environment. *IEEE Trans. Comput.*, 39(4):447–459, 1990. http://dx.doi.org/10.1109/12.54838.

[19] Reinhard Schwarz and Friedemann Mattern. Detecting causal relationships in distributed computations: In search of the holy grail. *Distrib. Comput.*, 7(3):149–174, March 1994. https://doi.org/10.1007/BF02277859.

[20] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. A comprehensive study of Convergent and Commutative Replicated Data Types. Technical Report 7506, Institut National de la Recherche en Informatique et Automatique (INRIA), Rocquencourt, France, January 2011. http://hal.archives-ouvertes.fr/inria-00555588/.

[21] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In *Stabilization, Safety, and Security of Distributed Systems (SSS'11)*, volume 6976 of *LNCS*, pages 386–400. Springer, 2011. http://dx.doi.org/10.1007/978-3-642-24550-3_29.

[22] Douglas B. Terry, Marvin M. Theimer, Karin Peterson, Alan J. Demers, Mike J. Spreitzer, and Carl H. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *Mobility*, pages 322–334. ACM, 1999. http://dl.acm.org/citation.cfm?id=303461.342780.

[23] Robert H. Thomas. A majority consensus approach to concurrency control for multiple copy databases. *ACM Trans. Database Syst.*, 4(2):180–209, 1979. http://doi.acm.org/10.1145/320071.320076.

[24] Todd Hoff. How League of Legends Scaled Chat to 70 Million Players - It takes a lot of Minions, Retrieved 22-Dec-2015. http://highscalability.com/blog/2014/10/13/how-league-of-legends-scaled-chat-to-70-million-players-it-t.html.

[25] Werner Vogels. Eventually consistent. *Queue*, 6(6):14–19, 2008. http://doi.acm.org/10.1145/1466443.1466448.

[26] Marek Zawirski, Carlos Baquero, Annette Bieniusa, Nuno Preguiça, and Marc Shapiro. Eventually consistent register revisited. In *Proceedings of the 2nd Workshop on the Principles and Practice of Consistency for Distributed Data (PaPoC'16)*, pages 9:1–9:3. ACM, 2016. http://doi.acm.org/10.1145/2911151.2911157.