Lasp: A Language for Distributed, Coordination-Free Programming

Christopher Meiklejohn
Basho Technologies, Inc.
cmeiklejohn@basho.com

Peter Van Roy
Université catholique de Louvain
peter.vanroy@uclouvain.be

Abstract

We propose Lasp, a new programming model designed to simplify large-scale distributed programming. Lasp combines ideas from deterministic dataflow programming together with conflict-free replicated data types (CRDTs). This provides support for computations where not all participants are online together at a given moment. The initial design presented here provides powerful primitives for composing CRDTs, which lets us write long-lived fault-tolerant distributed applications with nonmonotonic behavior in a monotonic framework. Given reasonable models of node-to-node communications and node failures, we prove formally that a Lasp program can be considered as a functional program that supports functional reasoning and programming techniques. We have implemented Lasp as an Erlang library built on top of the Riak Core distributed systems framework. We have developed one nontrivial large-scale application, the advertisement counter scenario from the SyncFree research project. We plan to extend our current prototype into a general-purpose language in which synchronization is used as little as possible.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming; E.1 [*Data Structures*]: Distributed data structures

Keywords Eventual Consistency, Commutative Operations, Erlang

1. Introduction

Synchronization of data across systems is becoming increasingly expensive and impractical when running at the scale required by "Internet of Things" [29] applications and large online mobile games.[1] Not only does the time required to coordinate with an ever growing number of clients increase with each additional client, but techniques that rely on coordination of shared state, such as Paxos and state-machine replication, grow in complexity with partial replication, dynamic membership, and unreliable networks. [14]

[1] Rovio, developer of the popular "Angry Birds" game franchise reported that during the month of December 2012 they had 263 million active users. This does not account for users who play the game on multiple devices, which is an even larger number of devices requiring some form of shared state in the form of statistics, metrics, or leaderboards. [3]

PPDP '15, July 14 - 16, 2015, Siena, Italy.
Copyright © 2015 ACM 978-1-4503-3516-4/15/07...$15.00.
http://dx.doi.org/10.1145/2790449.2790525

This is further complicated by an additional requirement for both of these applications: each must tolerate periods without connectivity while allowing local copies of replicated state to change. For example, mobile games should allow players to continue to accumulate achievements or edit their profile while they are riding in the subway without connectivity; "Internet of Things" applications should be able to aggregate statistics from a power meter during a snowstorm when connectivity is not available, and later synchronize when connectivity is restored. Because of these requirements, the burden is placed on the programmer of these applications to ensure that concurrent operations performed on replicated data have both a deterministic and desirable outcome.

For example, consider the case where a user's gaming profile is replicated between two mobile devices. Concurrent operations, which can be thought of as operations performed during the period where both clients are online but without communication, can modify the same state: the burden is placed on the application developer to write application logic that resolves these conflicting updates. This is true even if the changes commute: for instance, concurrent modifications to the user profile where client A modifies the profile photo and client B modifies the profile's e-mail address.

Recently, a formalism has been proposed by Shapiro et al. for supporting deterministic resolution of individual objects that are acted upon concurrently in a distributed system. These data types, referred to as Conflict-Free Replicated Data Types (CRDTs), provide a property formalized as Strong Eventual Consistency: given all updates to an object are eventually delivered in a distributed system, all copies of that object will converge to the same state. [32, 33]

Strong Eventual Consistency (SEC) results in deterministic resolution of concurrent updates to replicated state. This property is highly desirable in a distributed system because it no longer places the resolution logic in the hands of the programmer; programmers are able to use replicated data types that function as if they were their sequential counterparts. However, it has been shown that arbitrary composition of these data types is nontrivial. [6, 13, 15, 26]

To achieve this goal, we propose a novel programming model aimed at simplifying correct, large-scale, distributed programming, called Lasp.[2] This model provides the ability to use operations from functional programming to deterministically compose CRDTs into larger computations that observe the SEC property; these applications support programming with data structures whose values appear nonmonotonic externally, while computing internally with the objects' monotonic metadata. This model builds on our previous work, Derflow and Derflow$_L$ [12, 27], which provide a distributed,

[2] Inspired by LISP's etymology of "LISt Processing", our fundamental data structure is a join-semilattice, hence Lasp.

fault-tolerant variable store powering a deterministic concurrency programming model.

This paper has the following contributions:

- **Formal semantics:** We provide the formal semantics for Lasp: the monotonic **read** operation; functional programming operations over sets, including **map**, **filter**, and **fold**; and set-theoretic operations, including **product**, **union**, and **intersection**.
- **Formal theorem:** We formally prove that a distributed execution of a Lasp program can be considered a functional program that supports functional reasoning and programming techniques.
- **Prototype implementation:** We provide a prototype implementation [1] of Lasp, implemented as an Erlang library using the Riak Core [22] distributed systems framework.
- **Initial evaluation:** We perform an initial evaluation of Lasp by prototyping the eventually consistent advertisement counter scenario from the SyncFree project [4] and improve on the design of the Bloom KVS presented by Conway et al. [15]

This paper is an extension of the previously published work-in-progress report on Lasp [28] and is structured as follows: Section 2 motivates the need for Lasp. Section 3 defines Lasp's semantics and operations. Section 4 defines and proves the fundamental theorem of Lasp execution. Section 5 explains the implementation of our prototype. Section 6 evaluates the expressiveness of our prototype by showing how to implement two nontrivial distributed applications. Section 7 explains how Lasp is related to other work on models and languages for distributed programming. Section 8 outlines the extensions that we are planning for the Lasp prototype. Section 9 gives a brief conclusion.

2. Motivation

In this section, we motivate the need for Lasp.

2.1 Conflict-Free Replicated Data Types

Conflict-Free Replicated Data Types [32, 33] (CRDTs) are distributed data types that are designed to support temporary divergence at each replica, while guaranteeing that once all updates are delivered to all replicas of a given object they will converge to the same state. There are CRDT designs for commonly used sequential data types: counters, registers, sets, flags, dictionaries, and graphs; however, each of these data structures, while guaranteeing to converge, will observe a predetermined bias on how to handle concurrent operations, given that behavior is undefined in its sequential counterpart.

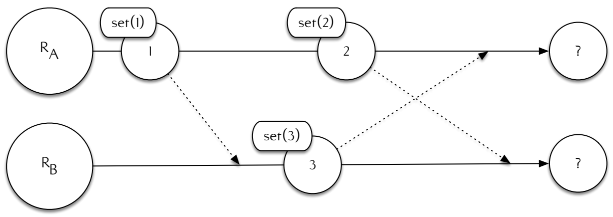

Figure 1: Example of divergence due to concurrent operations on replicas of the same object. In this case, it is unclear which update should win when replicas eventually communicate with each other.

We provide an example in Figure 1. In this example, concurrent operations on a replicated register result in divergence at each replica: replica A (R_A) is set to the value 2 whereas replica B (R_B) is set to the value 3. How do we reconcile this divergence?

Two major strategies have been used in practice by several production databases [18, 25]: "Last-Writer-Wins", where the last object written based on wall clock time wins, or "Semantic Resolution" where both updates are stored at each replica, and the burden of resolving the divergence is placed on the developer.

Both of these strategies have deficiencies:

- "Last-Writer-Wins" ultimately results in some valid operations being dropped during merge operations based solely on the scheduling of operations.
- "Semantic Resolution" places the burden on the application developer to provide a deterministic merge function.[3]

CRDTs solve this problem by formalizing a series of data types that fulfill two major goals: capturing causal information about updates that have contributed to their current state, and providing a deterministic merge operation for combining the state across multiple replicas.

Figure 2 shows one possible way to define a merge function that is deterministic regardless of ordering of messages: if we take advantage of the order of natural numbers using the *max* operation, we ensure that all replicas will eventually converge to a correct, equivalent state once they have observed all updates. While this is one very simple example of a distributed data structure with a deterministic merge function, Shapiro et al. outline different designs for registers, sets, counters, and graphs [33].

We will now formalize the Observed-Remove Set CRDT to explore problems of composition with CRDTs that have visible nonmonotonic behavior.

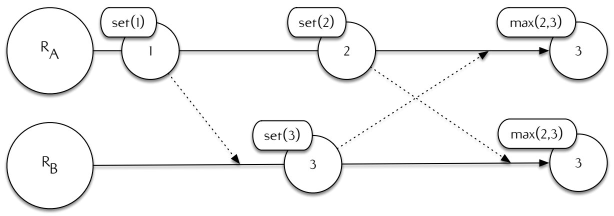

Figure 2: Example of resolving concurrent operations with a type of state-based CRDT based on a natural number lattice where the join operation computes max.

2.2 Observed-Remove Set CRDT

We take a moment to introduce the Observed-Remove Set CRDT (OR-Set), which will be used as the basis for the formalisms in this paper. We focus on the OR-Set because it is the least complex CRDT which serves as a general building block for applications.[4]

We start with a description of lattices, which are used as the basis of state-based CRDTs, one of the two major types of CRDTs.

Definition 2.1. A **bounded join-semilattice** is a partially ordered set that has a binary operation called the *join*. The *join* is associative, commutative, and idempotent, and induces a partial order over a nonempty finite subset such that the result given any two elements

[3] When described by DeCandia et al. in 2007 [18], this mechanism results in "concurrency anomalies", where updates seem to reappear due to concurrent operations in the network; this is the main focus of the the CRDT work as presented by Shapiro et al. in [32, 33].

[4] For instance, the Grow-Only Set (G-Set) does not allow removals, the Two-Phase Set (2P-Set) only allows one removal of a given item, and the OR-Set Without Tombstones (ORSWOT) adds additional complexity in the form of optimizations, which lie outside of the core semantics.

is the least upper bound of the input with respect to the partial order. The semilattice is bounded, as it contains a least element. [16]

Definition 2.2. A replicated triple (S, M, Q) where S is a bounded join-semilattice representing the state of each replica, M is a set of functions for mutating the state, and Q is a set of functions for querying the state, is one type of **state-based CRDT**. [5]

Functions for querying and mutating the CRDT can always be performed as they are executed on the replica's local state and the entire state is propagated to other replicas in the replica set. When a replica receives a state from another replica, the received state is merged into the local state using the $join$ operation. Given the algebraic properties of the $join$ operation, once updates stop being issued, a join across all replicas produces equivalent state; the merge function is deterministic given a finite set of updates.

Mutations at a given replica always return a monotonically greater state as defined by the partial order of the lattice, therefore any subsequent state always subsumes a previous state. We refer to these mutations as **inflations**.

Definition 2.3. A **stream** s is an infinite sequence of states of which only a finite prefix of length n is known at any given time.

$$s = [s_i \mid i \in \mathbb{N}] \tag{1}$$

The execution of one CRDT replica is represented by a stream of states, each of which is an element of the lattice S. The execution of a full CRDT instance with n replicas is represented by n streams.

Definition 2.4. Updates performed on a given state are **inflations**; for any mutation, the state generated from the mutation will always be strictly greater than the state generated by the previous mutation.

$$m \in M \land s_i \in S \land s_i \sqsubseteq m(s_i) \tag{2}$$

The **Observed-Remove Set** CRDT models arbitrary nonmonotonic operations, such as additions and removals of the same element, monotonically in order to guarantee convergence with concurrent operations at different replicas.

Definition 2.5. The **Observed-Remove Set (OR-Set)** is a state-based CRDT whose bounded join-semilattice is defined by a set of triples, where each triple has one value v, and extra information (called *metadata*) in the form of an add set a and a remove set r. At most one triple may exist for each possible value of v.

$$s_i = \{(v, a, r), (v', a', r'), \ldots\} \tag{3}$$

The OR-Set has two mutations, **add** and **remove**. The metadata is used to implement both mutations monotonically.

Definition 2.6. The **add** function on an OR-Set generates a unique constant u for each invocation. Given this constant, add u to the add set a for value v, if the value already exists in the set, or add a new triple containing v, an add set $\{u\}$ and a remove set $\{\}$.

$$\begin{aligned} add(s_i, v) = s_i - \{(v, \_, \_)\} \\ \cup \{(v, a \cup \{u\}, r) \mid (v, a, r) \in s_i \land u = unique()\} \\ \cup \{(v, \{u\}, \{\}) \mid \neg(v, \_, \_) \in s_i \land u = unique()\} \end{aligned} \tag{4}$$

Definition 2.7. The **remove** function on an OR-Set for value v unions all values in the add set for value v into the remove set for value v.

$$remove(s_i, v) = s_i - \{(v, \_, \_)\} \cup \{(v, a, a \cup r) \mid (v, a, r) \in s_i\} \tag{5}$$

The OR-Set has one query function that returns the current contents of the set: **query**.

Definition 2.8. The **query** function for an OR-Set returns values which are currently present in the set. Presence of a value v in a given state s_i is determined by comparison of the remove set r with the add set a. If the remove set r is a proper subset of the add set a, the value v is present in the set.

$$query(s_i) = \{v \mid (v, a, r) \in s_i \land r \subset a\} \tag{6}$$

The Observed-Remove Set is one instance of a CRDT that has a **query** function that is nonmonotonic: the data structure allows arbitrary additions and removals of elements in a set. It is important to distinguish between the external representation of the set (the output of a query, which is nonmonotonic) and the internal representation (the result of add and remove operations, which are monotonic).

2.3 Composition

The convergence properties of CRDTs are highly desirable for computation in distributed systems: these data structures are resilient to update reordering, duplication, and message delays, all of which are very relevant problems for computation on an unreliable asynchronous network. However, these convergence properties only hold for individual replicated objects and do not extend to computations that compose more than one CRDT.

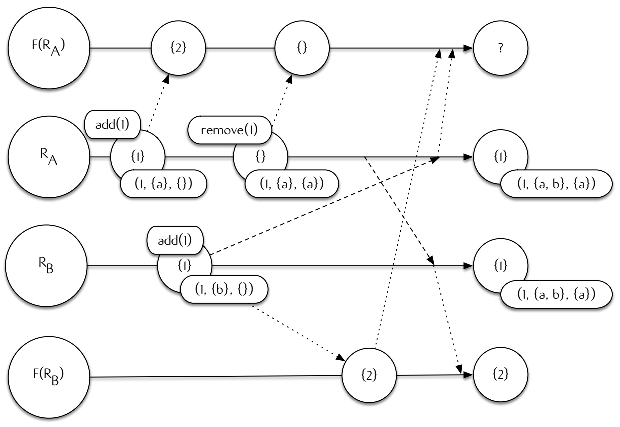

Figure 3: *Example of CRDT composition. In this example, there are two replicas of a CRDT, R_A and R_B; a function F, defined as $\lambda x.2x$, is applied to each element in the set at each replica using the **map** function. Without properly mapping the metadata, the convergence property does not hold for the result of the function application.*

In Figure 3, we see an example where the internal state of both replicas A and B (R_A and R_B) allows us to reason about state that reflects visible nonmonotonic behavior, additions and removals of the same element, by modeling the state changes monotonically. However, if we apply a function to the external representation of the value then we sacrifice the convergence property.

This example describes a case where the output of a function F, defined as $\lambda x.2x$, applied to each element at replica A ($F(R_A)$) using the **map** function receives state from the same function applied to the elements at replica B ($F(R_B)$). It is unclear how to merge the incoming state given we can not determine if the incoming state has been previously observed or not.[5]

3. Lasp

We now present the API and semantics of Lasp, a programming model designed for building convergent computations by composing CRDTs.

[5] Technically, in this naive mapping the state and value are the same.

3.1 API

Lasp's programming model is provided as a library in the Erlang programming language. This library implements the core semantics of Lasp and provides a distributed runtime for executing Lasp applications. The primary data type of Lasp is the CRDT. Given a CRDT instance of type t, the Lasp API is designed as follows:

Core API Core functions are responsible for defining variables, setting their values and reading the result of variable assignments.

- $\mathit{declare}(t)$: Declare a variable of type t.[6]
- $\mathit{bind}(x, v)$: Assign value v to variable x. If the current value of x is w, this assigns the join of v and w to x.
- $\mathit{update}(x, \mathrm{op}, a)$: Apply op to x identified by constant a.
- $\mathit{read}(x, v)$: Monotonic read operation; this operation does not return until the value of x is greater than or equal to v in the partial order relation induced over x at which time the operation returns the current value of x.
- $\mathit{strict\_read}(x, v)$: Same as $\mathit{read}(x, v)$ except that it waits until the value of x is strictly greater than v.

Functional Programming API Functional programming primitives define processes that never terminate; each process is responsible for reading subsequent values of the input and writing to the output. Figure 4 shows use of the **map** function.

- $\mathit{map}(x, f, y)$: Apply function f over x into y.
- $\mathit{filter}(x, p, y)$: Apply filter predicate p over x into y.
- $\mathit{fold}(x, \mathrm{op}, y)$: Fold values from x into y using operation op.

Set-Theoretic API Set-theoretic functions define processes that never terminate; each process is responsible for reading subsequent values of the input and writing to the output.

- $\mathit{product}(x, y, z)$: Compute product of x and y into z.
- $\mathit{union}(x, y, z)$: Compute union of x and y into z.
- $\mathit{intersection}(x, y, z)$: Compute intersection of x and y into z.

3.2 Processes

The previously introduced Lasp operations, functional and set-theoretic, create processes that connect all replicas of two or more CRDTs. Each process tracks the monotonic growth of the internal state at each replica and maintains a functional semantics between the state of the input and output instances. Each process correctly transforms the internal metadata of the input CRDTs to compute the correct mapping of value and metadata for the output CRDT.[7] For example, the Lasp **map** operation can be used to connect two instances of the Observed-Remove Set CRDT.

In the **map** example seen in Figure 4, whenever an element e is added or removed from the input set, the mapped version $f(e)$ is correctly added or removed from the output set. The other operations provided by Lasp are analogous: the user visible behavior is the normal result of the functional or set-theoretic function.

3.3 Variables

As we will prove in Section 4, each state-based CRDT in Lasp has the appearance of a single state sequence that evolves monotonically over time as update operations are issued; this is similar to the definition of inflation provided earlier (Definition 2.4). The current state of the CRDT is stored in a variable; successive values of the variable form the CRDT's state sequence.

We now formally define variables in Lasp and invariants the Lasp system preserves for each variable.

[6] Given the Erlang programming library does not have a rich type system, it is required to declare CRDTs with an explicit type at initialization time.

[7] The internal metadata of each CRDT is responsible for ensuring correct convergence; the transformation is therefore required to be deterministic.

```
%% Create initial set S1.
{ok, S1} = lasp:declare(riak_dt_orset),

%% Add elements to initial set S1 and update.
{ok, _} = lasp:update(S1, {add_all, [1,2,3]}, a),

%% Create second set S2.
{ok, S2} = lasp:declare(riak_dt_orset),

%% Apply map operation between S1 and S2.
{ok, _} = lasp:map(S1, fun(X) -> X * 2 end, S2).
```

Figure 4: *Map function applied to an OR-Set using the Erlang API of our Lasp prototype. We ignore the return values of the functions, given the brevity of the example.*

3.4 Monotonic Read

The *monotonic read* operation ensures that read operations always read an equivalent or greater value when provided with the result of a previous read. This behavior is very important to our system when dealing with replicated data to ensure forward progress. Consider the following example:

- Variable a is replicated three times, on three nodes: a_1, a_2, a_3.
- Application reads variable a from replica a_1.
- Application modifies replica a_1; state is then asynchronously propagated to replicas a_2 and a_3.
- Application reads variable a from replica a_2, because replica a_1 is temporarily unreachable.

In this example, it is possible for replica a_2 to temporarily have previous state than replica a_1, given message delays, failures, and asynchronous replication.[8] The *monotonic read* operation ensures that the read will not complete until an equivalent or greater state as defined over the partial order for a's lattice is available at a given replica based on a *trigger* value.

Formally, we define the *monotonic read* operations as follows:

Definition 3.1. The **monotonic read** operation defines a process that reads the known elements of the input stream s and waits until some s_i is equal to or monotonically greater than s_e. At this point, s_i is returned.

$$\begin{aligned} \mathit{read}(s, s_e) = \exists i.\ & s_i \in s \wedge s_e \sqsubseteq s_i \\ & [t_j \mid t_j = (j \geq i \Rightarrow s_i; \bot)] \end{aligned} \tag{7}$$

We also provide a strict version of the *monotonic read* operation, which does not return until a strict inflation of a previous read has been observed. This allows us to build recursive functions, such as our functional programming operations and set-theoretic operations, in terms of tail-recursive processes which continuously observe increasing state.

We define the strict version of the *monotonic read* operation as follows:

[8] This is a core idea behind eventual consistency and replication strategies such as optimistic replication. Eventually consistent systems ensure updates are eventually visible (i.e., in finite time), but make no guarantees about when the updates will be visible. [18, 31]

Definition 3.2. The **monotonic strict read** operation defines a process that reads the known elements of the input stream s and waits until some s_i is monotonically greater than s_e. At this point, s_i is returned.[9]

$$\begin{aligned} \mathit{strict\_read}(s, s_e) = \exists i.\ s_i \in s \wedge s_e \sqsubset s_i \\ [t_j \mid t_j = (j \geq i \Rightarrow s_i; \bot)] \end{aligned} \tag{8}$$

3.5 Functional Programming

We now look at the semantics for functional programming primitives that are lifted to operate over CRDTs: **map**, **filter**, and **fold**. We formalize them as follows:

Definition 3.3. The **map** function defines a process that never terminates, which reads elements of the input stream s and creates elements in the output stream t. For each element, the value v is separated from the metadata, the function f is applied to the value, and new metadata is attached to the resulting value $f(v)$. If two or more values map to the same $f(v)$ (for instance, if the function provided to map is surjective), the metadata is combined into one triple for all values of v.

$$\begin{aligned} F(s_i, f) &= \{f(v) \mid (v, \_, \_) \in s_i\} \\ A(s_i, f, w) &= \bigcup \{a \mid (v, a, \_) \in s_i \wedge w = f(v)\} \\ R(s_i, f, w) &= \bigcup \{r \mid (v, \_, r) \in s_i \wedge w = f(v)\} \\ \mathit{map}'(s_i, f) &= \{(w, A(s_i, f, w), R(s_i, f, w)) \mid w \in F(s_i, f)\} \\ \mathit{map}(s, f) &= t = [\mathit{map}'(s_i, f) \mid s_i \in s] \end{aligned} \tag{9}$$

Figure 4 provides an example of applying the **map** function to an OR-Set. In this example, the user does not need to know the internal data structure of each CRDT, but only the nonmonotonic external representation, as the Lasp runtime handles the metadata mapping automatically.

Definition 3.4. The **filter** function defines a process that never terminates, which reads elements of the input stream s and creates elements in the output stream t. Values for which $p(v)$ does not hold are removed by a metadata computation, to ensure that the filter is a monotonic process.

$$\begin{aligned} \mathit{filter}'(s_i, p) &= \{(v, a, r) \mid (v, a, r) \in s_i \wedge p(v)\} \\ &\quad \cup \{(v, a, a \cup r) \mid (v, a, r) \in s_i \wedge \neg p(v)\} \\ \mathit{filter}(s, p) &= t = [\mathit{filter}'(s_i, p) \mid s_i \in s] \end{aligned} \tag{10}$$

Definition 3.5. The **fold** function defines a process that never terminates, which reads elements of the input stream s and creates elements in the output stream t. Given $\mathit{query}(s_i) = V = \{v_0, ..., v_{n-1}\}$ and an operation op of t's type with neutral element e, this should return the state $t_i = e \text{ op } v_0 \text{ op } v_1 \ \cdots \text{ op } v_{n-1}$. If $\mathit{remove}(v_k)$ is done on s_i, then v_k is removed from V, so v_k must be removed from this expression in order to calculate t_{i+1}. The difficulty is that this must be done through a monotonic update of t_i's metadata. We present a correct but inefficient solution below; we are actively working on more efficient solutions.

$$\begin{aligned} \mathit{fold}'(s_i, \text{op}) &= \text{Op}_{(v,a,r) \in s_i}(\text{Op}_{u \in a} v \ \text{ op } \ \text{Op}'_{u \in r} v) \\ \mathit{fold}(s, \text{op}) &= t = [\mathit{fold}'(s_i, f) \mid s_i \in s] \end{aligned} \tag{11}$$

This solution assumes that op is associative, commutative, and has an inverse denoted by op$'$. Note that the elements u of a are not used directly; they serve only to ensure that the operation op(v) is repeated $|a|$ times (and analogously for op$'(v)$ which is repeated $|r|$ times). Since a and r grow monotonically, it is clear that the computation of $\mathit{fold}'(s_i, \text{op})$ also grows monotonically.

[9] This waits for a strict inflation in the lattice, as opposed to an inflation, which triggers when the value does not change.

3.6 Set-Theoretic Functions

We now look at the semantics for the set-theoretic functions that are lifted to operate over CRDTs: **product**, **union**, and **intersection**. We formalize them as follows:

Definition 3.6. The **product** function defines a process that never terminates, which reads elements of the input streams s and u, and creates elements in the output stream t. A new element is created on t for each new element read on either s and u. Metadata composition ensures that if v_s is removed from s or v_u is removed from u, then all pairs containing v_s or v_u are removed from t.[10]

$$\begin{aligned} \mathit{product}'(s_i, u_j) = \{((v, v'), a \times a', a \times r' \cup r \times a') \\ \mid (v, a, r) \in s_i, (v', a', r') \in u_j\} \\ \mathit{product}(s, u) = t = [\mathit{product}'(s_i, u_j) \mid s_i \in s, u_j \in u] \end{aligned} \tag{12}$$

Definition 3.7. The **union** function defines a process that never terminates, which reads elements of the input streams s and u, and creates elements in the output stream t. A new element is created on t for each new element read on either s or u. We combine the metadata for elements that exist in both inputs, similar to the definition of the **map** operation.

$$\begin{aligned} \mathit{un}_1(s_i, u_j) &= \{(v, a, r) \mid (v, a, r) \in s_i \oplus (v, a, r) \in u_j\} \\ \mathit{un}_2(s_i, u_j) &= \{(v, a \cup a', r \cup r') \mid (v, a, r) \in s_i, (v, a', r') \in u_j\} \\ \mathit{union}(s, u) &= t = [\mathit{un}_1(s_i, u_j) \cup \mathit{un}_2(s_i, u_j) \mid s_i \in s, u_j \in u] \end{aligned} \tag{13}$$

Definition 3.8. The **intersection** function defines a process that never terminates, which reads elements of the input streams s and u, and creates elements in the output stream t. A new element is created on t for each new element read on either s and u. We combine the metadata such that only elements that are in both s and u appear in the output.

$$\begin{aligned} \mathit{inter}'(s_i, u_j) = \{(v, a \times a', a \times r' \cup r \times a') \\ \mid (v, a, r) \in s_i, (v, a', r') \in u_j\} \\ \mathit{intersection}(s, u) = t = [\mathit{inter}'(s_i, u_j) \mid s_i \in s, u_j \in u] \end{aligned} \tag{14}$$

4. Fundamental Theorem of Lasp Execution

How easy is programming in Lasp? Can it be as easy as programming in a non-distributed language? Is it possible to ignore the replica-to-replica communication and distribution of CRDTs? Because of the strong semantic properties of CRDTs, it turns out that this is indeed possible. In this section we formalize the distributed execution of a Lasp program and we prove that there is a centralized execution, i.e., a single sequence of states, that produces the same result as the distributed execution. This allows us to use the same reasoning and programming techniques as centralized programs.

The programmer can reason about instances of CRDTs as monotonic data structures linked by monotonic functions, which is a form of deterministic dataflow programming. It has the good properties of functional programming (e.g., confluence and referential transparency) in a concurrent setting.[11]

4.1 Formal Definition of a CRDT Instance

We provide a formal definition of a CRDT instance and its distributed execution. For reasons of clarity, we borrow the notations of the original report on CRDTs [32].

[10] When $r = a$ or $r' = a'$ then $a \times r' \cup r \times a' = a \times a'$, and when $r \subset a$ and $r' \subset a'$ then $a \times r' \cup r \times a' \subset a \times a'$.

[11] See chapter 4 of [34] for a detailed presentation of deterministic dataflow.

Notation for Replication and Method Executions Assume a replicated object with n replicas and one state per replica. We use the notation s_i^k for the state of replica i after k method executions. The vector $(s_0^{k_0}, \cdots, s_{n-1}^{k_{n-1}})$ of the states of all replicas is called the object's *configuration*. A state is computed from the previous state by a method execution, which can be either an update or a merge. We have $s_i^k = s_i^{k-1} \circ f_i^k(a)$ where $f_i^k(a)$ is the k-th method execution at replica i. An update is an external operation on the data structure. A merge is an operation between two replicas that transfers state from one to another. A method execution that is an update is denoted $u_i^k(a)$ (it updates replica i with argument a). A method execution that is a merge is denoted $m_i^k(s_{i'}^{k'})$ (where $i \neq i'$; it merges state $s_{i'}^{k'}$ into replica i).

Definition 4.1. Causal order of method executions Method executions $f_i^k(a)$ have a causal order $\leq_H$ (H for *happens before*) defined by the following three rules:

1. $f_i^k(a) \leq_H f_i^{k'}(a')$ for all $k \leq k'$ (causal order at each replica)
2. $f_{i'}^{k'}(a) \leq_H m_i^k(s_{i'}^{k'})$ (causal order of replica-to-replica merge)
3. $f_i^k(a) \leq_H f_{i'}^{k'}(a')$ if there exists $f_{i''}^{k''}(a'')$ such that $f_i^k(a) \leq_H f_{i''}^{k''}(a'')$ and $f_{i''}^{k''}(a'') \leq_H f_{i'}^{k'}(a')$ (transitivity)

Definition 4.2. Delivery Using causal order we define the concept of delivery: an update $u_i^k(a)$ is *delivered* to a replica i at state $s_{i'}^{k'}$ if $u_i^k(a) \leq_H f_{i'}^{k'}(a)$.

Definition 4.3. State-based CRDT A CRDT is a replicated object that satisfies the following conditions:

- **Basic structure:** It consists of n replicas where each replica has an initial state, a current state, and two methods query and update that each executes at a single replica.
- **Eventual delivery:** An update delivered at some correct replica is eventually delivered at all correct replicas.
- **Termination:** All method executions terminate.
- **Strong Eventual Consistency (SEC):** All correct replicas that have delivered the same updates have equal state.

This definition is slightly more general than the definition of report [32]. In that report, an additional condition is added: that each replica will always eventually send its state to each other replica, where it is merged using a join operation. We consider that this condition is too strong, since there are many ways to ensure that state is disseminated among the replicas so that eventual delivery and strong eventual consistency are guaranteed. In its place, we assume a weak synchronization model, Property 4.2, that is not part of the CRDT definition, and we allow each CRDT to send the merge messages it requires to satisfy the CRDT properties.

Theorem 4.1. Monotonic semilattice condition for CRDTs *A replicated object is a **state-based CRDT instance** (in short, a CRDT instance), if the following three conditions hold:*

1. *The set of possible values of a state s_i^k forms a semilattice ordered by $\sqsubseteq$.*
2. *Merging state s with state s' computes the Least Upper Bound (join) of the two states $s \circ s'$.*
3. *The state is monotonically non-decreasing across updates: $s \sqsubseteq s \circ u$ for any update u.*

*We say that any CRDT instance satisfying this theorem is a **monotonic semilattice object**.*

Proof Proof is given in [32].

Definition 4.4. SEC state From the commutativity and associativity of the join operator $\circ$, it follows that for any execution of a monotonic semilattice object, if updates $U = \{u_0, ..., u_{n-1}\}$ are all delivered in state s, then $(u_0 \circ u_1 \circ \cdots \circ u_{n-1}) \sqsubseteq s$, that is, s is an inflation of the join of all updates in U. It is not necessarily equal since other updates may have occurred during the execution. We call $(u_0 \circ u_1 \circ \cdots \circ u_{n-1})$ the *SEC state* of updates U.

4.2 Formal Definition of a Lasp Process

We provide a formal definition of a Lasp process.

Definition 4.5. Monotonic m-ary function Given an m-ary function f between states such that $s = f(s_0, s_1, \cdots, s_{m-1})$. Then f is a *monotonic function* if $\forall i : \ s_i \sqsubseteq s_i' \Rightarrow f(\cdots, s_i, \cdots) \sqsubseteq f(\cdots, s_i', \cdots)$.

Definition 4.6. Lasp process A *Lasp process* is a pair of a sequence of m CRDT instances and one monotonic m-ary function f, written as $([C_0, \cdots, C_{m-1}], f)$. The process defines its output as n states where each state is the result of applying f on the corresponding replicas of the input CRDTs.

4.3 System Properties

The following properties are needed to prove the fundamental theorem.

Property 4.1. Fault model and repair We assume the following three conditions:

- **Crash-stop failures:** replicas fail by crashing and any replica may fail at any time.
- **Anti-entropy:** after every crash, a fresh replica is eventually created with state copied from any correct replica.
- **Correctness:** at least one replica is correct at any instant.

The first condition is imposed by the environment. The second condition is the repair action done by every CRDT when one of its replicas crashes. The third condition is what must hold globally for the CRDT to continue operating correctly.

Property 4.2. Weak synchronization For any execution of a CRDT instance, it is always true that eventually every replica will successfully send a message to each other replica.[12]

Property 4.3. Determinism Given two executions of a CRDT instance with the same sequence of updates but a different merge schedule, i.e., a different sequence of replica-to-replica communication, replicas in the first execution that have delivered the same updates as replicas in the second execution have equal state.

Since we intend Lasp programming to be similar to functional programming, it is important that computations are deterministic. We remark that SEC by itself is not enough to guarantee that; we provide a simple counterexample with the OR-Set:

Assume that replica A (R_A) does an $add(1)$ followed by a $remove(1)$ and replica B (R_B) does an $add(1)$. When all replicas have delivered these three updates, the state of the OR-Set will either contain 1 or not contain 1. It will not contain 1 if the second $add(1)$ is in the causal history of the $remove(1)$. In the other case, it will contain 1. Both situations are possible depending on whether or not the merge schedule communicates the state of replica B to replica A after its $add(1)$ and before the $remove(1)$. Figure 5 illustrates this scenario.

Therefore, to correctly use an OR-Set in Lasp, it is important to impose conditions that ensure determinism. The following two conditions are sufficient to guarantee determinism for all merge schedules:

[12] The content of this message depends on the definition of the CRDT.

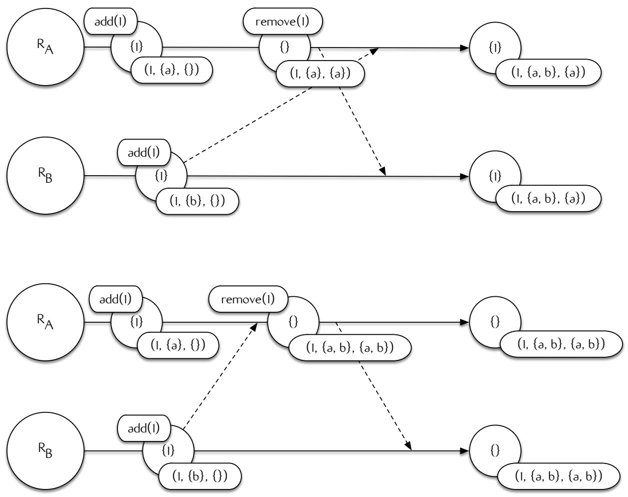

Figure 5: *Example of nondeterminism introduced by different replica-to-replica merge schedules. In the top example, merging after the remove results in the item remaining in the set, where merging before the remove results in the item being removed.*

- A $remove(v)$ is only allowed if an $add(v)$ with the same value has been done previously at the same replica.
- An $add(v)$ with the same value of v may not be done at two different replicas.

4.4 Lemmas

Lemma 4.2. Eventual delivery for faulty execution *Each update in a CRDT instance execution that satisfies Property 4.1 and Property 4.2, and for which the messages in Property 4.2 are delivered according to a continuous probability distribution is eventually delivered at all replicas or at no replicas, with probability 1.*

Proof According to Property 4.1, a replica may crash and be replaced by a new replica with state copied from any live replica. In any configuration of a CRDT execution, there will be $m \leq n$ live replicas of which $m' \leq m$ have delivered the update. Initially when the update is done, $m' = 1$. Crash of a replica that has delivered the update will decrease m'. Replica-to-replica communication will increase m' if done from a replica that has delivered the update to a replica that has not. Otherwise it will not affect m'. As the CRDT instance continues its execution, Property 4.1 implies that one of two situations will eventually happen: either all live replicas deliver the update, or no live replicas deliver the update. Once one of these situations happens, the third condition of Property 4.1 ensures it will continue indefinitely.

The continuous probability distribution ensures that all infinite non-converging executions have probability zero. For example, given three replicas R_A, R_B, R_C, and only R_C has delivered. It is possible that R_A crashes just after R_C delivers to it, followed by a new replica $R_{A'}$ created from R_B. This can repeat indefinitely while satisfying Property 4.1 and Property 4.2. With a continuous probability distribution, each repetition multiplies the probability by a number less than 1, so the infinite execution has probability zero.

Definition 4.7. Compatibility Given a CRDT instance execution and a finite set U of updates in this execution. We say that a state is *compatible* with U in the CRDT execution if it consists of the join of all updates in U inflated with any subset of the other updates occurring before the state.

Compatibility makes precise the notion that all replicas reach the same state if no other updates occur (SEC) but that other updates might occur in the meantime. All the replica states are not necessarily the same, but they are all inflations of the SEC state.

Lemma 4.3. Reduction of CRDT execution to a single state execution *For any CRDT instance execution, there exists a single state execution such that any finite set U of updates from the CRDT execution is eventually delivered to the single state execution and gives a state that is compatible with U in the CRDT execution.*

Proof Define a single state execution whose updates are a topological sort of the updates in the CRDT execution that respects the causal order $\leq_H$ of these updates. The resulting execution satisfies all four properties of Definition 4.3. In particular, for eventual delivery, it is clear that the single state execution eventually delivers all updates in U. All other updates occurring before this state are either causally before or concurrent with an update in U.

Lemma 4.4. Reduction of Lasp process to a CRDT execution *A Lasp process behaves as if it were a single CRDT instance with n replicas. Each replica state consists of an $(m+1)$-vector of the m states of the input CRDT instances and the corresponding state of the output as defined by f applied to the m input states.*

Proof The execution of the Lasp process satisfies all four properties of Definition 4.3. In particular, for strong eventual consistency is clear that each CRDT instance will eventually deliver its updates to all its replicas, resulting in a state compatible with these updates. When this happens for all CRDT instances, then all n replicas of the output state will be equal.

4.5 Fundamental Theorem

We present the fundamental theorem of Lasp.

Definition 4.8. Simple Lasp program A simple Lasp program consists of either:

- A single CRDT instance, or
- A Lasp process with m inputs that are simple Lasp programs and one output CRDT instance.

Theorem 4.5. *A simple Lasp program can be reduced to a single state execution.*

Proof This is straightforward to prove by induction. We construct the program in steps starting from single CRDTs. By Lemma 4.3, a single CRDT instance can be reduced to a single state execution. For each Lasp process, we replace it by a single CRDT instance whose updates are the updates of all its input CRDT instances. By Lemma 4.4, this is correct. We continue until we have constructed the whole program. By Lemma 4.2, if there are faults then the worst that can happen is that some updates are ignored.

5. Implementation

Our prototype of Lasp is implemented as an Erlang library. We leverage the *riak_dt* [2] library from Basho Technologies, Inc., which provides an implementation of state-based CRDTs in Erlang.

5.1 Distribution

Lasp distributes data using the Riak Core distributed systems framework [22], which is based on the Dynamo system [18].

Riak Core The Riak Core library provides a framework for building applications in the style of the original Dynamo system. Riak Core provides library functions for cluster management, dynamic membership and failure detection.

Dynamo-style Partitioning and Hashing Lasp uses Dynamo-style partitioning of CRDTs: consistent hashing and hash-space partitioning are used to distribute copies of CRDTs across nodes in a cluster to ensure high availability and fault tolerance. Replication of each CRDT is performed between adjacent nodes in a cluster. While the partitioning mechanism and implementation is nuanced, it is sufficient to realize the collection of CRDTs as a series of disjoint replica sets, of which the data is sharded across, with full replication between the nodes in any given replica set.

Anti-Entropy Protocol We provide an active anti-entropy protocol built on top of Riak Core that is responsible for ensuring all replicas are up-to-date. Periodically, a process is used to notify replicas that contain CRDT replicas with the value of a CRDT from a neighboring replica.[13]

Quorum System Operations In Section 4, we outline the three properties of our system: crash-stop failures, anti-entropy, and correctness. While these properties are sufficient to ensure confluence of computations, they do not guarantee that all updates will be observed if a given replica of a CRDT fails before communicating its state to a peer replica. Therefore, to guarantee safety and be tolerant to failures, both read and update operations are performed against a quorum of replicas. This ensures fault tolerance: by performing read and write operations against a majority, the system is tolerant to failures. The system remains safe and does not make progress when the majority is not available. Additionally, quorum operations can be used to increase liveness in the system: by writing back the merged value of the majority, we can passively repair objects during normal system operation, improving anti-entropy.[14]

Replication and Execution of Operations Given replication of the objects themselves, to ensure fault-tolerance and high-availability, our functional programming operations and set-theoretic operations must be replicated as well. To achieve this, quorum replication is used to contact a majority of replicas near the output CRDT, which are responsible for reading the input CRDT and performing the transformation.

Given the **map** example in Figure 4, we spawn processes at a majority of the output CRDT replicas, *S2*, which read from the input replicas of *S1*.

To ensure forward progress of these computations, each of our operations uses the strict version of the *monotonic read* operation to prevent from executing over stale values when talking to replicas which are out-of-date. In the **map** example, the transformation is performed for a given observation in the stream of updates to variable *S1* with the output written into the stream for variable *S2*, at which the process tail-recursively executes and wait to observe a causally greater value than the previously observed *S1* before proceeding. This prevents duplication of already computed work and ensure forward progress at each replica.

Additionally, we can apply read repair and anti-entropy techniques to repair the value of *S2* if it falls very far behind instead of relying on applying operations from *S1* in order.

6. Evaluation

In this section, we look at two applications that can be implemented with Lasp.

[13] We plan to design an optimized version, similar to the Merkle tree based approach in [18]; our current protocol is sufficient to ensure progress.

[14] In [18], this process is referred to as read repair.

6.1 Advertisement Counter

One of the use cases for our model is supporting clients that need to operate without connectivity. For example, imagine a provider of mobile games that sells advertisement space within their games.

In this example, the correctness criteria are twofold:

- Clients will go offline: consider mobile devices such as cellular phones that experience periods without connectivity. When the client is offline, advertisements should still be displayable.
- Advertisements need to be displayed a minimum number of times. Additional impressions are not problematic.

Figure 6 presents one design for an eventually consistent advertisement counter written in Lasp. In this example, squares represent primitive CRDTs and circles represent CRDTs that are maintained using Lasp operations. Additionally, Lasp operations are represented as diamonds and edges represent the monotonic flow of information.

Our advertisement counter operates as follows:

- Advertisement counters are grouped by vendor.
- All advertisement groups are combined into one list of advertisements using a **union** operation.
- Advertisements are joined with active "contracts" into a list of displayable advertisements using both the **product** and **filter** operations.
- Each client selects an advertisement to display from the list of active advertisements.
- For each advertisement displayed, each client updates its local copy of the advertisement counter.
- Periodically, advertisement counters are merged upstream.
- When a counter hits at least 50,000 advertisement impressions, the advertisement is "disabled" by removing it from the list of advertisements.

The implementation of this advertisement counter is completely monotonic and synchronization-free. Adding and removing ads, adding and removing contracts, and disabling ads when their contractual number of views is achieved are all modeled as the monotonic growth of state in CRDTs connected by active processes. Programmer-visible nonmonotonicity is represented by monotonic metadata in the CRDTs.

The full implementation of the advertisement counter is available in the Lasp source code repository and consists of 213 LOC. In this example, transparent distribution and failure handling is supported by the runtime environment, and not exposed to the developer. For brevity, we provide only two code samples: the advertisement counter "server" process, that is responsible for disable advertisements when their threshold is reached, and example use of the **product** and **filter** operations used for composing the advertisements with their contracts.

Figure 7 provides an example of the advertisement counters server process: this process is responsible for performing a blocking read on each counter that will disable the counter by removing it from the set once the threshold is reached. One server is launched per counter to manage its lifecycle.

Figure 8 provides an example of the advertisement counters dataflow: both of the **product** and **filter** operations spawn processes that continuously compute the composition of both the set of advertisements and the set of counters as each data structure independently evolves.

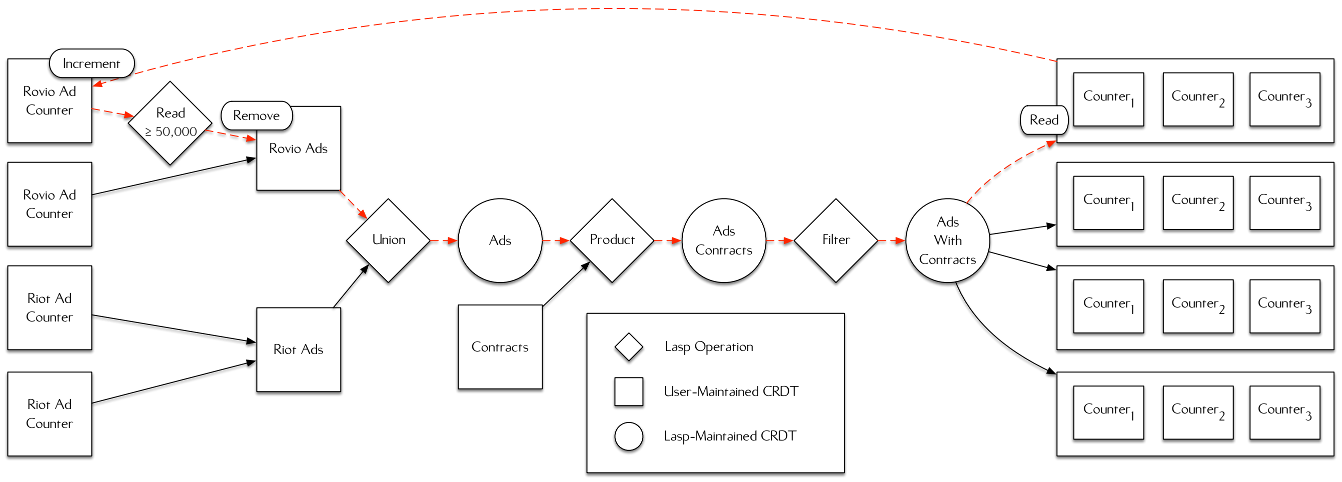

Figure 6: *Eventually consistent advertisement counter. The dotted line represents the monotonic flow of information for one counter.*

```
%% @doc Server for the advertisement counter.
server({#ad{counter=Counter}=Ad, _}, Ads) ->
    %% Blocking monotonic read for 50,000
    {ok, _} = lasp:read(Counter, 50000),

    %% Remove the advertisement.
    {ok, _} = lasp:update(Ads, {remove, Ad}, Ad),

    lager:info("Removing ad: ~p", [Ad]).
```

Figure 7: *Example use of the monotonic read operation in the advertisement counter application. A process is spawned that blocks until the advertisement counter reaches 50,000 impressions, after which it removes itself from the list of advertisements.*

```
    %% Compute the Cartesian product of both
    %% ads and contracts.
    {ok, AdsContracts} = lasp:declare(?SET),
    ok = lasp:product(Ads, Contracts, AdsContracts),

    %% Filter items by join.
    {ok, AdsWithContracts} = lasp:declare(?SET),
    FilterFun = fun({#ad{id=Id1},
                       #contract{id=Id2}}) ->
        Id1 =:= Id2
    end,
    ok = lasp:filter(AdsContracts,
                       FilterFun,
                       AdsWithContracts),
```

Figure 8: *Example use of dataflow operations in the advertisement counter application. These operations together compute a join between a set of advertisements and a set of counters to compute a list of displayable advertisements.*

6.2 Bloom^L Replicated Key-Value Store

We provide an example of a replicated key-value store (KVS) similar to the key-value store example presented by Conway et al. [15]. In this example, we show how our model supports writing this replica in a easy to reason about functional manner.

Our key-value store is a simple recursive function that receives three types of messages from clients: *get*, *put*, and *remove*.

- *get:* Retrieve a value from the KVS by name.
- *put:* Store a value in the KVS by name, computing the *join* of the new value and the current value.
- *remove:* Remove observed values in the KVS by key.

Figure 9 contains the code for a single server replica. A *riak_dt_map*, a composable, convergent map, [13] is used for modeling the store. Given this data structure supports the composition of state-based CRDTs, we assume the values for all keys will be mergeable given the lattice defined by the data type stored.

```
receiver(Map, ReplicaId) ->
    receive
        {get, Key, Client} ->
            {ok, {_, MapValue0, _}} = lasp:read(Map),
            MapValue = riak_dt_map:value(MapValue0),
            case orddict:find(Key, MapValue) of
                error ->
                    Client ! {ok, not_found};
                Value ->
                    Client ! {ok, Value}
            end,
            receiver(Map, ReplicaId);
        {put, Key, Value, Client} ->
            {ok, _} = lasp:update(Map,
                                  {update,
                                   [{update, Key,
                                    {add, Value}}]},
                                  ReplicaId),
            Client ! ok,
            receiver(Map, ReplicaId);
        {remove, Key, Client} ->
            {ok, _} = lasp:update(Map,
                                  {update,
                                   [{remove, Key}]},
                                  ReplicaId),
            Client ! ok,
            receiver(Map, ReplicaId)
    end.
```

Figure 9: *Simple replicated key-value store in Lasp. This tail-recursive process is responsible for receiving messages from client processes, and processes them in serial order.*

In our example, we use a simple recursive process for modeling the key-value store. This process is responsible for responding to both *get* and *put* messages: when a message is received the appropriate action is performed on the given key. When a *put* message arrives, the map is updated by performing two actions: first, merging the current value with the provided value in the map, second, merging the updated map back into the variable store with the new map. This operation is done atomically by Lasp using the *update* operation. When a *get* message arrives, we return the current value from the map for the provided key. Multiple instances of the replicated KVS can merge state by periodic exchange of their maps.

We improve on the Bloom^L KVS by supporting concurrent removal operations: removals observed at a replica remove all observed values for a key while concurrent additions for the same key win against concurrent removals. Lasp's programming model removes the restrictions placed on lattices having external monotonic behavior by using CRDTs as the primary programming abstraction, while additionally providing a familiar functional programming semantics to simplify distributed programming.

7. Related Work

In the following section, we identify related work.

7.1 Distributed Oz

Distributed Oz [19, 34] provides an extension of the Oz programming model allowing for asynchronous communication and mobile processing. Distributed Oz formalizes this by extending the Oz centralized execution semantics with semantics for distributed execution. Distributed Oz has a functional core that performs distributed unification over rational trees [35]. For unifications without conflicting bindings, this satisfies the definition of a CRDT.

Lasp's use of CRDTs solves the problem of conflicting bindings: for each type of CRDT, there is always a merge function that can resolve concurrent operations in a deterministic manner. In addition, Lasp provides deterministic dataflow over general CRDTs, whereas Distributed Oz provides deterministic dataflow over just one CRDT, namely rational trees. Finally, Lasp uses metadata computation to support nonmonotonic operations in a functional setting.

7.2 FlowPools

FlowPools [30] provide a lock-free deterministic concurrent dataflow abstraction for the Scala programming language. FlowPools are essentially a lock-free collection abstraction that support a concurrent append operation and a set of combinators and higher-order operations. FlowPools are designed for multi-threaded computation, not distributed computation.

While higher-order operations such as **foreach** and **aggregate** function similarly to the **map** and **fold** operations in Lasp, namely they execute once for each element that will eventually exist in the FlowPool, these operations are somewhat limited. Each FlowPool can only be appended to, and each element is single-assignment. Computations using the **aggregate** operation require that the FlowPool be sealed before the result of the aggregation is realized.

7.3 Derflow and Derflow$_L$

Derflow and Derflow$_L$ are direct precursors to Lasp. Derflow [12] defines a fault-tolerant single-assignment data store. It implements deterministic dataflow programming [34] on the Dynamo-inspired, Riak Core distributed systems framework. [18, 22]

Derflow$_L$ [27] extends Derflow to join-semilattices. Derflow$_L$ relies on user-specified composition of CRDTs. While this model is sufficient for composition of less complex CRDTs, it fails to scale to the more complex and efficient CRDTs since it requires the programmer to explicitly handle the composition of metadata.

7.4 Bloom^L

Bloom^L [15] provides Datalog-style operations over monotonically growing lattices in a distributed environment. Applications in Bloom^L can be analyzed to identify locations where nonmonotonic operations occur, where coordination can be used to enforce order. Differences in the programming abstraction notwithstanding, we highlight two differences between Bloom^L and Lasp:

Retraction of Information Retraction of information in Bloom^L is nonmonotonic, and therefore not confluent. By using composition of OR-Sets, Lasp can offer an eventually consistent (monotonic and confluent) mechanism for the retraction of information, but can not guarantee when the update might be visible.

Sealing Lattices used by the Bloom^L system lack causal information, which places the requirement on monotone functions to, once satisfied, freeze, or seal, their values. [7]

For instance, consider the case of a monotonic mapping between two booleans, a to b: once a becomes true, b becomes true. Once the condition is met in a, and b is set to true; the property is considered "satisfied" and can no longer become "unsatisfied". This prevents the situation where an earlier version of an update is delivered to the system and prevents the condition from observing nonmonotonic behavior. Lasp can detect these scenarios using metadata in the form of logical clocks which can be tracked through morphisms, preventing an earlier update from causing the regression of the value.

7.5 LVars

LVars [24] formalizes lattice variables for use in parallel computations in single machine settings that enforce determinism. While LVars shares a similar functional programming core with Lasp, each system differs in its distribution and failure modes given they were designed to solve different problems. We also believe the threshold read operation formalized in the LVars work is insufficient for use with advanced types of CRDTs.

We discuss both of these issues below:

Differences in Design: LVars vs. Lasp Focusing on single machine computations, LVars is aimed at running computations in parallel, over shared state, while preserving determinism in an otherwise functionally pure application. Focusing on distributed computations, Lasp is aimed at running fault-tolerant applications, which are designed to diverge and later converge deterministically, given periods of time where processes may not be able to communicate.

Threshold Reads vs. Monotonic Reads The *threshold read* operation, both originally formalized over lattices and later reformalized over state-based CRDTs[15] [23] by Kuper and Newton, makes two assumptions: *a priori* knowledge of the internal state of a CRDT to properly threshold on the value, and that the queryable value of a CRDT is monotone.

For example, the Grow-Only Set CRDT (G-Set) observes both of these properties: when writing a deterministic computation over a known stream, you are able to satisfy both conditions. First, the internal representation of the CRDT requires only storage of the set structure itself; concurrent operations can potentially add the same element, however, the *join* operation between two sets will remove duplicates and advance the data structure in the partial order. Second, the value of the set will always be increasing and therefore is monotone.

However, when dealing with a design like the Observed-Remove Set CRDT (OR-Set), which allows the repeated addition

[15] The citation refers to these as "CvRDTs", convergent replicated data types, a legacy name for the more recent nomenclature, "state-based."

and removal of arbitrary elements, individual operations on the data structure must be uniquely identified for correct semantics, even if they represent the concurrent addition of the same element at two different replicas. This prevents the first property from being fulfilled: *a priori* knowledge of the unique constant identifiers given multiple executions of the same program under different interleavings is not possible. The second property cannot be fulfilled either: the Observed-Remove Set has a nonmonotonic query function because elements can be removed.

Lasp is designed to specifically address the two previously discussed problems: the problem of *threshold reads* given CRDTs with nondeterministic internal state, and the problem of properly composing these data types, while preserving the internal knowledge required for correct convergence.

7.6 Discretized Streams

Discretized Streams (D-Streams) [37] is a programming model for stream processing that supports efficient parallel recovery of faults. D-Streams realize infinite streams as a series of small immutable batches over which deterministic computations can be done, as typically seen in the MapReduce model. [17] The major contribution of this work is efficient parallel recovery during faults; instead of replicating the computations of the streams or using upstream backup, [8, 20] lost computations can be recomputed in parallel.

The model exploits the immutable nature of the individual events in infinite streams; D-Streams assume that a batch is considered sealed at a given time and events are grouped into batches based on when the event arrives at the ingestion point. On the other hand, Lasp assumes that individual data structures, along with compositions of these data structures, will monotonically evolve over time while preserving determinism.

7.7 Summingbird

Summingbird [11] is an open-source domain specific language for integrating online and batch computations into a single programming abstraction. Summingbird can be used to build complex DAG workflows, where processing is performed between sources and sinks, with the additional ability to persist both partial and final results to a data store such as MySQL or HBase.

In enabling correct, efficient aggregation of computations, operations in the "reduce" phase are restricted to commutative semigroups. This prevents incorrect operation in the event of network or processing anomalies such as out-of-order message delivery.

Lasp's primary programming abstraction is the state-based CRDT: a convergent data structure formalized with a bounded join-semilattice. Given a semilattice is a commutative idempotent semigroup, and a bounded join-semilattice forms a commutative idempotent monoid which induces a partial order using the join operation, this allows Lasp to handle both the network anomalies of duplicated and reordered messages, as well as to reason about the ordering of updates to a given item.

8. Current and Future Work

In the following section, we identify current and future work.

8.1 General Concepts

Currently, Lasp is a first-order model that allows defining data structures and operations performed on them. Future extensions will add abstraction mechanisms and other concepts, as they are needed by the application scenarios that we intend to implement. The concepts will be designed according to the needs of expressiveness and efficiency explained in the following two sections.

8.2 Invariant Preservation

Some computations require the preservation of invariants between sets of replicated CRDTs. One such example is the students and teams example posed by Conway et al. [15] when discussing the "scope problem" of CRDTs. In this scenario, removing a student from the set of active students should also remove the student from any teams they were participating in.

We envision a way to specify these invariants between CRDTs as contracts: these contracts would be enforced by a mechanism at runtime given an allowed amount of divergence. We look at two examples of where contracts would be useful:

- In the advertisement counter example (Figure 6), clients can locally increment their counter, and either synchronize with the server side advertisement counter for a given advertisement after every impression or after a given number of impressions. How often a client chooses to synchronize is a measure of how much we allow this counter to diverge.
- In the students and teams example, we may want to enforce the invariant locally at each replica, but allow the system to temporarily diverge to reduce the amount of synchronization.

We believe we can leverage recent work in contract enforcement and invariant preservation in eventually consistent systems, specifically "invariant-based programming" by Balegas et al. [9] and Quelea by Kaki et al. [21].

8.3 Optimizations

We plan to explore optimizing the Lasp programming model through the use of metadata reduction, reduced state propagation, and intermediate tree elimination.

Metadata Reduction Lasp currently has limited support for the Optimized Conflict-Free Replicated Set (ORSWOT) as described by Bieniusa et al. [10]. This set contains a novel algorithm for avoiding the requirement of tracking tombstones (i.e., it does not need garbage collection of metadata). This makes it an ideal data structure for use in production systems. This data structure is also the basis for the convergent, conflict-free replicated map, as described by Brown et al. [13].

Reduced State Propagation We would like to explore methods for reducing the amount of computation needed to propagate state changes through the graph. Two such approaches for this are Operation-based CRDTs [33], which propagate commutative operations through a reliable channel instead of the full state, and $\delta-$state CRDTs [5], which propagate minimal state representing the delta derived by applying the operation locally.

Distributed Intermediate Tree Elimination Computations in Lasp are formed using a very small subset of a functional language: this results in very large tree structures, where the intermediate computations might not be necessary. This is a side effect of the functional programming style. We imagine that techniques such as Wadler's "deforestation" can be used to eliminate these structures in a distributed fashion for more efficient computation, resulting in less network communication. [36]

9. Conclusions

We introduced the Lasp programming model and motivated its use for large-scale computation over replicated data. Our future plans for Lasp include extending it to become a full-fledged language and system, identifying optimizations for more efficient state propagation, exploring stronger consistency models, and optimizing distribution and replica placement for better fault tolerance and reduced latency. We also plan to evaluate the Lasp system and to test our hypothesis that Lasp's weak synchronization model is well-suited for scalable and high-performance applications, in particular in settings with intermittent connectivity such as mobile applications and "Internet of Things". Our ultimate goal is for Lasp to become a general purpose language for building large-scale distributed applications in which synchronization is used as little as possible.

Acknowledgments

Thanks to Peter Alvaro, Sean Cribbs, Scott Lystig Fritchie, Seyed (Hossein) Haeri, Joel Knighton, John Muellerleile, Andrew Stone, and our anonymous reviewers for their feedback. The research leading to these results has been partially funded by the SyncFree Project [4] in the European Union Seventh Framework Programme (FP7/2007-2013) under grant agreement n^o 609551.

References

[1] Lasp source code repository. `https://github.com/lasp-lang/lasp`. Accessed: 2015-06-14.

[2] Riak DT source code repository. `https://github.com/basho/riak_dt`. Accessed: 2015-03-30.

[3] 263 Million Monthly Active Users In December. `http://www.rovio.com/en/news/blog/261/263-million-monthly-active-users-in-december/`. Accessed: 2015-02-13.

[4] SyncFree: Large-scale computation without synchronisation. `https://syncfree.lip6.fr`. Accessed: 2015-02-13.

[5] P. S. Almeida, A. Shoker, and C. Baquero. Efficient State-based CRDTs by Delta-Mutation. *arXiv preprint arXiv:1410.2803*, 2014.

[6] P. Alvaro, P. Bailis, N. Conway, and J. M. Hellerstein. Consistency without borders. In *Proceedings of the 4th annual Symposium on Cloud Computing*, page 23. ACM, 2013.

[7] P. Alvaro, N. Conway, J. M. Hellerstein, and D. Maier. Blazes: Coordination analysis for distributed programs. In *Data Engineering (ICDE), 2014 IEEE 30th International Conference on*, pages 52–63. IEEE, 2014.

[8] M. Balazinska, H. Balakrishnan, S. R. Madden, and M. Stonebraker. Fault-tolerance in the borealis distributed stream processing system. *ACM Transactions on Database Systems (TODS)*, 33(1):3, 2008.

[9] V. Balegas, M. Najafzadeh, S. Duarte, M. Shapiro, R. Rodrigo, and N. Preguiça. Putting Consistency Back into Eventual Consistency. *Submitted to EuroSys 2015*.

[10] A. Bieniusa, M. Zawirski, N. Preguiça, M. Shapiro, C. Baquero, V. Balegas, and S. Duarte. An optimized conflict-free replicated set. *arXiv preprint arXiv:1210.3368*, 2012.

[11] O. Boykin, S. Ritchie, I. OConnell, and J. Lin. Summingbird: A framework for integrating batch and online mapreduce computations. *Proceedings of the VLDB Endowment*, 7(13), 2014.

[12] M. Bravo, Z. Li, P. Van Roy, and C. Meiklejohn. Derflow: distributed deterministic dataflow programming for Erlang. In *Proceedings of the Thirteenth ACM SIGPLAN workshop on Erlang*, pages 51–60. ACM, 2014.

[13] R. Brown, S. Cribbs, C. Meiklejohn, and S. Elliott. Riak DT map: a composable, convergent replicated dictionary. In *Proceedings of the First Workshop on Principles and Practice of Eventual Consistency*, page 1. ACM, 2014.

[14] T. D. Chandra, R. Griesemer, and J. Redstone. Paxos made live: an engineering perspective. In *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, pages 398–407. ACM, 2007.

[15] N. Conway, W. R. Marczak, P. Alvaro, J. M. Hellerstein, and D. Maier. Logic and lattices for distributed programming. In *Proceedings of the Third ACM Symposium on Cloud Computing*, page 1. ACM, 2012.

[16] B. A. Davey and H. A. Priestley. *Introduction to lattices and order*. Cambridge University Press, 2002.

[17] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[18] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: amazon's highly available key-value store. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 205–220. ACM, 2007.

[19] S. Haridi, P. Van Roy, and G. Smolka. An overview of the design of Distributed Oz. In *Proceedings of the second international symposium on parallel symbolic computation*, pages 176–187. ACM, 1997.

[20] J.-H. Hwang, M. Balazinska, A. Rasin, U. Cetintemel, M. Stonebraker, and S. Zdonik. High-availability algorithms for distributed stream processing. In *Data Engineering, 2005. ICDE 2005. Proceedings. 21st International Conference on*, pages 779–790. IEEE, 2005.

[21] G. Kaki, K. Sivaramakrishnan, and S. Jagannathan. Declarative programming over eventually consistent data stores. *To appear at PLDI'15*.

[22] R. Klophaus. Riak core: building distributed applications without shared state. In *ACM SIGPLAN Commercial Users of Functional Programming*, page 14. ACM, 2010.

[23] L. Kuper and R. R. Newton. Joining forces. *Draft*.

[24] L. Kuper and R. R. Newton. LVars: lattice-based data structures for deterministic parallelism. In *Proceedings of the 2nd ACM SIGPLAN workshop on Functional high-performance computing*, pages 71–84. ACM, 2013.

[25] A. Lakshman and P. Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.

[26] C. Meiklejohn. On the composability of the Riak DT map: expanding from embedded to multi-key structures. In *Proceedings of the First Workshop on Principles and Practice of Eventual Consistency*, page 13. ACM, 2014.

[27] C. Meiklejohn. Eventual Consistency and Deterministic Dataflow Programming. *8th Workshop on Large-Scale Distributed Systems and Middleware*, 2014.

[28] C. Meiklejohn and P. Van Roy. Lasp: a language for distributed, eventually consistent computations with CRDTs. In *Proceedings of the First Workshop on Principles and Practice of Consistency for Distributed Data*, page 7. ACM, 2015.

[29] D. Miorandi, S. Sicari, F. De Pellegrini, and I. Chlamtac. Internet of things: Vision, applications and research challenges. *Ad Hoc Networks*, 10(7):1497–1516, 2012.

[30] A. Prokopec, H. Miller, T. Schlatter, P. Haller, and M. Odersky. Flowpools: A lock-free deterministic concurrent dataflow abstraction. In *Languages and Compilers for Parallel Computing*, pages 158–173. Springer, 2013.

[31] Y. Saito and M. Shapiro. Optimistic replication. *ACM Computing Surveys (CSUR)*, 37(1):42–81, 2005.

[32] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. Conflict-free replicated data types. Technical Report RR-7687, INRIA, 07 2011.

[33] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. A comprehensive study of convergent and commutative replicated data types. Technical Report RR-7506, INRIA, 01 2011.

[34] P. Van Roy and S. Haridi. *Concepts, techniques, and models of computer programming*. MIT Press, 2004.

[35] P. Van Roy, S. Haridi, P. Brand, G. Smolka, M. Mehl, and R. Scheidhauer. Efficient logic variables for distributed computing. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 21(3): 569–626, 1999.

[36] P. Wadler. Deforestation: Transforming programs to eliminate trees. In *ESOP'88*, pages 344–358. Springer, 1988.

[37] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 423–438. ACM, 2013.