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” applications and large online mobile games. 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]

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. 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, [12, 27], which provide a distributed, 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 pro-

---

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]

2 Inspired by LISP’s etymology of “LISt Processing”, our fundamental data structure is a join-semilattice, hence Lasp.
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.

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.

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 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

---

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.
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.

$$s = [s_i]_{i \in \mathbb{N}}$$ (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.

$$m \in M \land s_i \in S \land s_i \subseteq m(s_i)$$ (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 $v$.

$$s_i = \{(v, a, r), (v', a', r'), \ldots\}$$ (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 $\{\}$. The incoming state given we can not determine if the incoming state 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.

$$add(s_i, v) = s_i - \{(v, \_\_\_)\} \cup \{(v, a \cup \{u\}, r) | (v, a, r) \in s_i \land u = unique()\} \cup \{(v, \{u\}, \{\}) | -(v, \_\_\_) \in s_i \land u = unique()\}$$ (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 $v$.

$$remove(s_i, v) = s_i - \{(v, \_\_\_)\} \cup \{(v, a, a \cup r) | (v, a, r) \in s_i\}$$ (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.

$$query(s_i) = \{v | (v, a, r) \in s_i \land r \subset a\}$$ (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.

3. Lasp

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

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:

$$\lambda x.2x$$ (7)

Technically, in this naive mapping the state and value are the same.
Given the Erlang programming library does not have a rich type system, it is possible for reading subsequent values of the input and writing to the output. 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.

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

- `declare(t);` Declare a variable of type t.6
- `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.
- `update(x, op, a);` Apply op to x identified by constant a.
- `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.
- `strict_read(x, v);` Same as `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.

- `map(x, f, y);` Apply function f over x into y.
- `filter(x, p, y);` Apply filter predicate p over x into y.
- `fold(x, 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.

- `product(x, y, z);` Compute product of x and y into z.
- `union(x, y, z);` Compute union of x and y into z.
- `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. The 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.

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.

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: a₁, a₂, a₃.
- Application reads variable a from replica a₁.
- Application modifies replica a₁; state is then asynchronously propagated to replicas a₂ and a₃.
- Application reads variable a from replica a₂, because replica a₁ is temporarily unreachable.

In this example, it is possible for replica a₂ to temporarily have a previous state than replica a₁, given message delays, failures, and asynchronous replication.7 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 sᵢ is equal to or monotonically greater than sₑ. At this point, sᵢ is returned.

\[
\text{read}(s, sₑ) = \exists i. sᵢ \in s \land sᵢ \supseteq sₑ
\]

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:

**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 sᵢ is monotonically greater than sₑ. At this point, sᵢ is returned.8

\[
\text{strict_read}(s, sₑ) = \exists i. sᵢ \in s \land sᵢ \supset sₑ
\]

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.

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]
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.

\[ F(s, f) = \{ f(v) \mid (v, a, r) \in s \} \]

\[ A(s, f, w) = \bigcup \{ a \mid (v, a, r) \in s \land w = f(v) \} \]

\[ R(s, f, w) = \bigcup \{ r \mid (v, a, r) \in s \land w = f(v) \} \]

\[ \text{map}(s, f) = \{ (w, A(s, f, w), R(s, f, w)) \mid w \in F(s, f) \} \]

\[ \text{map}(s, f) = t = [\text{map}(s, f) \mid s_i \in s] \] \hspace{1cm} (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.

\[ \text{filter}'(s, p) = \{ (v, a, r) \mid (v, a, r) \in s_i \land p(v) \} \]

\[ \text{filter}(s, p) = t = [\text{filter}'(s, p) \mid s_i \in s] \] \hspace{1cm} (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 query(s) = V = \{v_0, \ldots, v_{n-1}\} and an operation op of t’s type with neutral element e, this should return the state \( t_i = e \cup v_0 \cup v_1 \cdots \cup v_{n-1} \). If 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.

\[ \text{fold}'(s, op) = \text{Op}(v, a, r) \subseteq s_i \cup \text{Op}_{v, a, r} \cup \text{Op}_{v, a, r} \]

\[ \text{fold}(s, op) = t = [\text{fold}'(s, op) \mid s_i \in s] \] \hspace{1cm} (11)

This solution assumes that op is associative, commutative, and has an inverse denoted by \( \text{op}^{-1} \). Note that the elements u of a are not used directly; they serve only to ensure that the operation \text{op}(v) is repeated \(|a|\) times (and analogously for \( \text{op}(v) \) which is repeated \(|r|\) times). Since a and r grow monotonically, it is clear that the computation of \( \text{fold}'(s, op) \) also grows monotonically.

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

\[ \text{product}'(s, u) = \{ (v, a \times a', r \times r') \mid v \in s_i \land u \in u_j \} \]

\[ \text{product}(s, u) = t = [\text{product}'(s, u) \mid s_i \in s, u_j \in u] \] \hspace{1cm} (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.

\[ \text{un}_1(s, u) = \{ (v, a, r) \mid (v, a, r) \in s_i \lor (v, a, r) \in u_j \} \]

\[ \text{un}_2(s, u) = \{ (v, a \cup a', r \cup r') \mid (v, a, r) \in s_i \lor (v, a', r') \in u_j \} \]

\[ \text{union}(s, u) = t = [\text{un}_1(s, u) \cup \text{un}_2(s, u) \mid s_i \in s, u_j \in u] \] \hspace{1cm} (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 or u. We combine the metadata such that only elements that are in both s and u appear in the output.

\[ \text{inter}'(s, u) = \{ (v, a \times a', r \times r') \mid (v, a, r) \in s_i \land (v, a', r') \in u_j \} \]

\[ \text{intersection}(s, u) = t = [\text{inter}'(s, u) \mid s_i \in s, u_j \in u] \] \hspace{1cm} (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.

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].

Notation for Replication and Method Executions Assume a replicated object with \( n \) replicas and one state per replica. We use the notation \( R_i \) for the state of replica \( i \) after \( k \) method executions. The vector \( \mathbf{s}_i = (s_0, \ldots, s_{n-1}) \) 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 \( s_i = s_{i-1} \circ f_i^k(a) \) 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.

\[ \text{product}'(s, u) = \left\{ (v, a \times a', r \times r') \mid v \in s_i \land u \in u_j \right\} \]

\[ \text{product}(s, u) = t = [\text{product}'(s, u) \mid s_i \in s, u_j \in u] \] \hspace{1cm} (12)

\[ \text{union}(s, u) = t = [\text{un}_1(s, u) \cup \text{un}_2(s, u) \mid s_i \in s, u_j \in u] \] \hspace{1cm} (13)

\[ \text{intersection}(s, u) = t = [\text{inter}'(s, u) \mid s_i \in s, u_j \in u] \] \hspace{1cm} (14)

11 See chapter 4 of [34] for a detailed presentation of deterministic dataflow.
that transfers state from one to another. A method execution that is an update is denoted $u_i^f(a)$ (it updates replica $i$ with argument $a$). A method execution that is a merge is denoted $m_i^f(s_i')$ (where $i \neq i'$; it merges state $s_i'$ 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. $f_i^k(a) \leq_H m_i^k(s_i')$ (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 $u_i^f(a)$ is delivered to a replica $i$ at state $s_i^f$ if $u_i^f(a) \leq_H f_i^f(a)$.

**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 $s_k$ with state $s_i'$ computes the Least Upper Bound (join) of the two states $s_k \sqcup s_i'$.  
3. The state is monotonically non-decreasing across updates: $s_k \sqsubseteq s_{k+1}$ for any update $u_i$.

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 $\sqcup$, it follows that for any execution of a monotonic semilattice object, if updates $U = \{u_0, ..., u_{n-1}\}$ are all delivered in state $s_k$, then $(u_0 \sqcup u_1 \sqcup ... \sqcup u_{n-1}) \sqsubseteq s$, 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 $(u_0 \sqcup u_1 \sqcup ... \sqcup u_{n-1})$ 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 $s = f(s_0, s_1, ..., s_{m-1})$. Then $f$ is a monotonic function if $\forall i: s_i \sqsubseteq s_i' \Rightarrow f(s_0, s_1, ..., s_{m-1}) \sqsubseteq f(s_0, s_1, ..., s_{m-1})$.

**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 $([C_0, ..., C_{m-1}], f)$. 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.

**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:

- 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

---

12 The content of this message depends on the definition of the CRDT.
There will be no new replicas with state copied from any live replica. 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 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_dlt [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 write 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, \(S_2\), which read from the input replicas of \(S_1\).

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 \(S_1\) with the output written into the stream for variable \(S_2\), at which the process tail-recursively executes and wait to observe a causally greater value than the previously observed \(S_1\) 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 \(S_2\) if it falls very far behind instead of relying on applying operations from \(S_1\) in order.

6. **Evaluation**

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

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 disabling 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.

6.2 **Bloom\(^{1}\) 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.

\(^{13}\)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.
% @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]).

% 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),

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

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.

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.

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.

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 LKVS by supporting concurrent removal operations: removals observed at a replica remove all ob-
erved 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 DerflowL
Derflow and DerflowL 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]

DerflowL [27] extends Derflow to join-semilattices. DerflowL relies on user-specifed 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 BloomL
BloomL [15] provides Datalog-style operations over monotonically growing lattices in a distributed environment. Applications in BloomL 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 BloomL and Lasp:

Retraction of Information Retraction of information in BloomL 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 BloomL 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 boonlas, \( a \) 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 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

15 The citation refers to these as “CvRDTs”, convergent replicated data types, a legacy name for the more recent nomenclature, “state-based.”
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 of faults during; 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 Bienuis 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 δ-state CRDTs [5], which propagate minimal state representing the delta derived by applying the operation locally.

Distributed Intermediate Tree Elimination Computations in Lasp are formulated 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° 609551.
References