Joining Forces

Toward a Unified Account of LVars and Convergent Replicated Data Types

Lindsey Kuper Ryan R. Newton

Indiana University

{lkuper, rrnewton}@cs.indiana.edu

Abstract

LVars—shared memory locations whose semantics are defined in terms of an application-specific lattice—offer a principled approach to deterministic-by-construction, shared-state parallel programming: writes to an LVar take the *join* of the old and new values with respect to the lattice, while reads from an LVar can observe only that its contents have crossed a specified "threshold" in the lattice. This interface guarantees that programs have a deterministic outcome, despite parallel execution and schedule nondeterminism.

LVars have a close cousin in the distributed systems literature: *convergent replicated data types* (CvRDTs), which leverage lattice properties to guarantee that all replicas of a distributed object (for instance, in a distributed database) are *eventually consistent*. Unlike LVars, in which all updates are joins, CvRDTs allow updates that are inflationary with respect to the lattice but do not compute a join. Moreover, CvRDTs differ from LVars in that they allow intermediate states to be observed: if two replicas of an object are updated independently, reads of those replicas may disagree until a (least-upper-bound) merge operation takes place.

Although CvRDTs and LVars were developed independently, LVars ensure determinism under parallel execution by leveraging the same lattice properties that CvRDTs use to ensure eventual consistency. Therefore, a sensible next research question is: how can we take inspiration from CvRDTs to improve the LVars model, and vice versa? In this paper, we take steps toward answering that question in both directions: we consider both how to extend CvRDTs with LVar-style threshold reads and how to extend LVars with CvRDT-style inflationary updates, and we advocate for the usefulness of these extensions.

1. Introduction

Deterministic-by-construction parallel programming models ensure that all programs written using the model have the same observable behavior every time they are run, offering freedom from subtle, hard-to-reproduce nondeterministic bugs in parallel code. Ideally, a deterministic-by-construction parallel program will run faster when more parallel resources are dynamically available, and so we do *not* necessarily want a deterministic-by-construction model to require that exact scheduling behavior is deterministic; only a program's *outcome* should be preserved across multiple runs. Indeed, we are interested in models that specifically *allow* tasks to be scheduled dynamically and unpredictably, in order to handle irregular parallel applications, without allowing such *schedule nondeterminism* to affect the outcome of a program.

In earlier work [8, 9], we proposed *LVars* as a principled approach to shared-state parallel programming with guaranteed observably deterministic outcomes. An LVar is a memory location that can be shared among multiple threads and accessed through `put` (write) and `get` (read) operations. Unlike a typical shared mutable location, though, the values an LVar can take on are elements of an application-specific *lattice*. This application-specific lattice determines the semantics of the `put` and `get` operations that comprise the interface to LVars:

- `put` operations can only change an LVar's state in a way that is *monotonically increasing* with respect to the application-specific lattice, because it updates the LVar to the *join*, or least upper bound, of the old state and the new state.
- `get` operations allow only limited observations of the state of an LVar. Every `get` operation must be expressible in terms of a *threshold set* of minimum values that can be read from the LVar, where every two elements in the threshold set must have the lattice's greatest element $\top$ as their join. A call to a `get` operation blocks until the LVar in question reaches a (unique) value in the threshold set, then unblocks and returns *that value*, rather than the LVar's exact contents.

Together, monotonically increasing writes via `put` and threshold reads via `get` yield a deterministic-by-construction programming model. That is, a program in which `put`s and `get`s on LVars are the only side effects will have the same observable result on every run, in spite of parallel execution and schedule nondeterminism [8].

Lattices for eventual consistency The problem of ensuring determinism of parallel programs is closely related to the problem of ensuring the *eventual consistency* [16] of replicated objects in a distributed system. Consider, for example, an object representing the contents of a shopping cart, replicated across a number of physical locations. If two replicas disagree on the contents of the cart—for instance, if one replica sees only that item a has been added to the cart, while another sees only item b—how do we know what the "real" cart contents are? One option is to give every write a timestamp and allow the last-written replica to overrule the others, but such a "last-write-wins" policy does not necessarily make sense from a semantic point of view [6]. In the particular case of the shopping cart, we might instead want to resolve the conflict by taking the set union $\{a, b\}$ of the two replicas' contents; for some other application, a different policy might be more appropriate.

This notion of application-specific conflict resolution, long used by, for instance, the Amazon Dynamo key-value store [6], has recently been formalized in the setting of *convergent replicated data types* (CvRDTs) [12, 13]. A CvRDT is a replicated object in which the states that replicas can take on can be viewed as elements of a join-semilattice. While at any given time, replicas may differ, conflicts between replicas can always be deterministically resolved by a *merge* operation that computes the join of the two replicas' states. As long as all replicas merge with one another periodically, eventual consistency is guaranteed.

Joining forces Although LVars and CvRDTs were developed independently, both models leverage the mathematical properties of

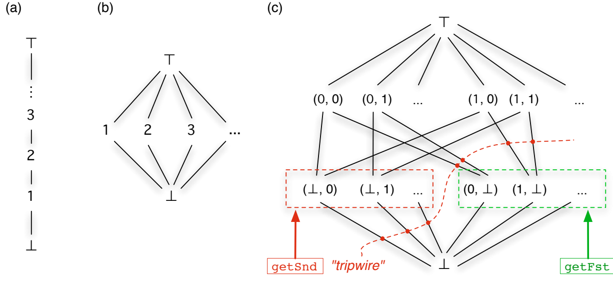

Figure 1: Example LVar lattices: (a) positive integers ordered by $\leq$; (b) IVar containing a positive integer; (c) pair of natural-number-valued IVars, annotated with example threshold sets that would correspond to a blocking read of the first or second element of the pair. Any state transition crossing the "tripwire" for `getSnd` causes it to unblock and return a result.

join-semilattices to ensure that a property of the model holds—determinism in the case of LVars; eventual consistency in the case of CvRDTs. Therefore, a sensible next research question is: what is the relationship between the two models, and can we take inspiration from the literature on CvRDTs to improve the LVars model, and vice versa? In this paper, after reviewing the basics of LVars (Section 2) and CvRDTs (Section 3), we approach that question in both directions:

- First (Section 4), we will consider how LVar-style threshold reads apply to the setting of CvRDTs. Since threshold reads guarantee that the order in which updates occur cannot be observed, they can, unlike ordinary CvRDT queries, prevent intermediate states of replicas from being observed. Therefore, threshold reads ensure a greater degree of consistency than ordinary, non-blocking queries, but at the price of read availability. We argue that this is a trade-off worth making.
- Second (Section 5), we will consider how CvRDT-style inflationary updates apply to the setting of LVars. In fact, as we will explain, inflationary non-join LVar updates are crucial for some LVar applications.

2. LVars Refresher

We start with a brief review of the LVars programming model. LVars are a mechanism for ensuring that shared-state parallel computations have deterministic outcomes by only allowing least-upper-bound writes and threshold reads of shared memory locations. In previous work [8, 9], we gave a formal treatment of LVars and proved determinism for LVar-based calculi. Here, we give a more informal introduction to LVars in the setting of *LVish*, our Haskell library for programming with LVars.[1] The example programs in this section are written using LVish and have types of the form `Par Det`, indicating that they run in the `Par` monad that the LVish library provides, with a deterministic (`Det`) *effect level*. In LVish programs, all operations that read and write LVars must run inside `Par` computations. [2]

LVars are a generalization of *IVars* [2], a well-known mechanism for deterministic parallel programming. An IVar is a write-once variable with a blocking read semantics: an attempt to read an empty IVar will block until the IVar has been filled with a value. LVars generalize IVars to allow multiple writes, so long as those writes are monotonically increasing with respect to an application-specific *lattice*[3] of states. As a simple first example, consider a program in which two parallel computations write to an LVar `lv`, with one thread writing the value `2` and the other writing `3`. Using LVish, one way to write such a program is as follows:

(Example 1)

```
import Data.LVar.MaxPosInt

p :: Par Det ()
p = do
  lv ← newMaxPosInt
  fork (put lv 2)
  fork (put lv 3)
  waitThresh lv 3
```

Although it is possible to define one's own LVar data types using LVish, Example 1 uses LVish's built-in `MaxPosInt` LVar data type. Two calls to `fork` launch asynchronous threads, each of which performs a `put` to an LVar `lv` of type `MaxPosInt`. `lv`'s lattice is the $\leq$ ordering on positive integers, as shown in Figure 1(a). `waitThresh` is an example of a `get` operation: it waits for the contents of `lv` to reach or surpass the "threshold" value of `3`, and returns (with a return value of `()`) once that point has been reached.

The two calls to `put` in Example 1 can run in arbitrary order, and they can run either before or after the call to `waitThresh`. Nevertheless, Example 1 is deterministic, since `put` takes the *join* of the old and new LVar contents. Since both `put`s must eventually run, `lv`'s contents will reach $\max(2, 3) = 3$ on every execution, since in `lv`'s lattice the join of two positive integers n_1 and n_2 is $\max(n_1, n_2)$. Therefore `waitThresh` will always eventually unblock, regardless of the order in which the `put`s occur.

Why is it necessary to take the join of the old and new values? If `put` were to update the contents of `lv` to the new value, we would have no guarantee of determinism: for instance, under a schedule in which `put lv 3` ran first, followed by `put lv 2` and then `waitThresh lv 3`, the call to `waitThresh` would block forever because `lv`'s contents would be `2` by the time it ran. But, since `put` takes the join of its argument and the current LVar contents, `put lv 2` has no effect when `lv`'s contents are already `3`, and so `waitThresh lv 3` can return immediately.

A lattice-generic model Although the lattice of Figure 1(a) is one possible ordering for the states of an LVar, the LVars model is *lattice-generic*: *any* choice of lattice will result in a deterministic outcome. For instance, consider a version of Example 1 in which we choose `lv`'s lattice to be that of Figure 1(b), in which the join of any two distinct positive integers is $\top$—that is, we want `lv` to have the behavior of an IVar. To get this behavior, rather than using the `MaxPosInt` type, we can use the `IVar` type also provided by LVish:

(Example 2)

```
import Data.LVar.IVar

p :: Par Det Int
p = do
  lv ← newIVar
  fork (put lv 2)
  fork (put lv 3)
  get lv
```

[1] LVish is available at `http://hackage.haskell.org/package/lvish`; the examples in this paper are written against release 1.1.2.

[2] The `Par` monad in LVish is a generalization of the original `Par` monad exposed by the *monad-par* library (`http://hackage.haskell.org/package/monad-par`). In real LVish programs, `Par` computations have a second phantom type parameter to ensure that LVars cannot be re-used in multiple `Par` computations, in a manner analogous to how Haskell's `ST` monad prevents an `STRef` from being returned from `runST`.

[3] Formally, the lattice of states is given as a 4-tuple $(D, \leq, \bot, \top)$ where D is a set, $\leq$ is a partial order on D, $\bot$ is D's least element according to $\leq$, and $\top$ is D's greatest element. We do *not* require that every pair of elements in D have a greatest lower bound, only a least upper bound; hence $(D, \leq, \bot, \top)$ is really a *bounded join-semilattice* with a designated greatest element ($\top$). For brevity, we use the term "lattice" as a shorthand.

Regardless of scheduling, Example 2 will deterministically raise an exception, indicating that conflicting writes to `lv` have occurred. Unlike with a traditional, single-write IVar, though, multiple writes of the *same* value (say, `put lv 2` and `put lv 2`) would *not* raise an exception, because the join of any positive integer and itself is that integer—corresponding to the fact that multiple writes of the same value do not allow any nondeterminism to be observed.

Threshold sets and incompatibility The LVars model guarantees determinism through a combination of least-upper-bound writes and *threshold reads*. In Example 1, the call `waitThresh lv 3` corresponds to querying the lattice of Figure 1(a) using the singleton threshold set $\{3\}$. Singleton threshold sets are the only valid threshold sets for `MaxPosInt` LVars, and hence the interface exposed by `waitThresh` takes a single integer argument. On the other hand, the `IVar` LVar in Example 2 provides a `get` operation corresponding to an infinite threshold set $\{1, 2, 3, ...\}$. This is a valid threshold set, since its elements are *pairwise incompatible* with respect to the lattice of Figure 1(b): every two distinct elements in the threshold set have $\top$ as their join.

As a final example of a threshold set, consider an LVar `lv` whose states form a lattice of *pairs* of natural-number-valued IVars; that is, `lv` contains a pair (m, n), where m and n both start as $\bot$ and may each be updated once with a non-$\bot$ value, which must be some natural number. This lattice is shown in Figure 1(c). An `IVarPair` LVar data type could expose `getFst` and `getSnd` operations for reading from the first and second entries of `lv`. For the call `getSnd lv`, the implicit threshold set is $\{(\bot, 0), (\bot, 1), \dots\}$, an infinite set. There is no risk of nondeterminism because the elements of the threshold set are pairwise incompatible with respect to `lv`'s lattice: informally, since the second entry of `lv` can only be written once, no more than one state from the set $\{(\bot, 0), (\bot, 1), \dots\}$ can ever be reached.

```
p :: Par Det Int                                    (Example 3)
p = do
  lv ← newPair
  fork (putFst lv 0)
  fork (putSnd lv 1)
  getSnd lv
```

In Example 3, `getSnd lv` may unblock and return `1` any time after the second entry of `lv` has been written, regardless of whether the first entry has been written yet. One way of visualizing the implicit threshold set of $\{(\bot, 0), (\bot, 1), \dots\}$ for `getSnd lv` is as a subset of edges in the lattice that, if crossed, allow the operation to unblock and return. Together these edges form a "tripwire". This visualization is pictured in Figure 1(c).

3. CvRDTs and Eventual Consistency

Distributed systems typically involve *replication* of data objects across a number of physical locations. Replication is of fundamental importance in such systems: it makes the system more robust to data loss and allows for good data locality. Given the importance and ubiquity of replication, it would be convenient if systems of distributed, replicated objects behaved indistinguishably from the more familiar programming model in which all data is on one machine and all computation takes place there. Unfortunately, this is not the case. The well-known *CAP theorem* [3, 7] of distributed computing imposes a trade-off between *consistency*, in which every replica sees the same information, and *availability*, in which all information is available for both reading and writing by all replicas.

Consistency and availability, though, are not binary properties; rather than having, for instance, either perfect availability or no availability at all, we can choose how much availability a system must have, then compromise on consistency as needed to achieve that level of availability. *Highly available* distributed systems, such as Amazon's Dynamo key-value store [6], give up on strong consistency in favor of *eventual consistency* [16], in which replicas may not always agree, but if updates stop arriving, all replicas will *eventually* come to agree.

Conflict resolution and CvRDTs How can eventually consistent systems ensure that all replicas of an object eventually come to agree? As a straw man proposal, we could vacuously satisfy the definition of eventual consistency by setting all replicas to some pre-determined value—but then, of course, we would lose all updates we had made to any of the replicas. A more practical proposal would be to try to determine which replica was written most recently, then declare that replica the winner. But this approach is also less than ideal: even if we had a way of reliably synchronizing clocks between replicas and could always determine which replica was written most recently, having the last write win might not make sense from a *semantic* point of view. The Dynamo developers acknowledge this in their discussion of application-specific mechanisms for resolving conflicts between replicas [6]:

> The next design choice is who performs the process of conflict resolution. This can be done by the data store or the application. If conflict resolution is done by the data store, its choices are rather limited. In such cases, the data store can only use simple policies, such as "last write wins", to resolve conflicting updates. On the other hand, since the application is aware of the data schema it can decide on the conflict resolution method that is best suited for its clients experience. For instance, the application that maintains customer shopping carts can choose to "merge" the conflicting versions and return a single unified shopping cart.

In other words, we can take advantage of the fact that, for a particular application, we know something about the meaning of the data we are storing, and then parameterize the data store by a pluggable, application-specific conflict resolution policy.

Implementing application-specific conflict resolution policies in an ad-hoc way for every application sounds tedious and error-prone.[4] Fortunately, we need not implement them in an ad-hoc way. Shapiro *et al.*'s *convergent replicated data types* (CvRDTs) [12, 13] provide a simple mathematical framework for reasoning about and enforcing the eventual consistency of replicated objects, based on viewing replica states as elements of a lattice and replica conflict resolution as that lattice's join operation. Next, we review the basics of CvRDTs from the work of Shapiro *et al.*, and then we discuss the relationship between CvRDTs and LVars.

Conditions for eventual consistency Shapiro *et al.* define an *eventually consistent* object as one that has the property of *convergence*: all correct replicas of the object to which the same updates have been delivered eventually have equivalent state. Convergence is one of three conditions that are required for eventual consistency; the other two are *eventual delivery*, meaning that all replicas receive all update messages, and *termination*, meaning that all method executions terminate (we discuss methods in more detail below).

Shapiro *et al.* further define a *strongly eventually consistent* (SEC) object as one that is eventually consistent and, in addition to being merely convergent, is *strongly convergent*, meaning that correct replicas to which the same updates have been delivered have equivalent state.[5] A *conflict-free replicated data type* (CRDT), then, is a data type (*i.e.*, a specification for an object) satisfying certain

[4] Indeed, as the developers of Dynamo have noted [6], Amazon's shopping cart presents an anomaly whereby removed items may re-appear in the cart!

[5] Strong eventual consistency is not to be confused with strong consistency: it is the combination of eventual consistency and strong convergence. Contrast with ordinary convergence, in which replicas only *eventually* have equivalent state. In a strongly convergent object, knowing that the same up-

conditions that are sufficient to guarantee that the object is SEC. (The term "CRDT" is used interchangeably to mean a specification for an object, or an object meeting that specification.)

There are two "styles" of specifying a CRDT: *state-based*, also known as *convergent*[6]; or *operation-based* (or "op-based"), also known as *commutative*. CRDTs specified in the state-based style are also called *convergent replicated data types*, abbreviated *CvRDTs*, while those specified in the op-based style are also called *commutative replicated data types*, abbreviated *CmRDTs*. Because it is based on the algebraic framework of join-semilattices, the state-based, CvRDT style is closer to the LVars model, and so it is CvRDTs that are our focus in this paper—although, as Shapiro *et al*. have shown, CmRDTs can emulate CvRDTs, and vice versa.

State-based objects Shapiro *et al*. specify a *state-based object* as a tuple (S, s^0, q, u, m), where S is a set of states, s^0 is the initial state, q is the *query method*, u is the *update method*, and m is the *merge method*. Objects are replicated across some finite number of processes, with one replica at each process. We assume that each replica begins in the initial state s^0. The state of a local replica may be queried via the method q and updated via the method u. Methods execute locally, at a single replica, but the merge method m can merge the state from a remote replica with the local replica; we assume that each replica sends its state to the other replicas infinitely often, and that eventually every update reaches every replica, whether directly or indirectly.

A *state-based* or *convergent* replicated data type (CvRDT) is a state-based object equipped with a partial order $\leq$, written as a tuple $(S, \leq, s^0, q, u, m)$, that has the following properties:

- S forms a join-semilattice ordered by $\leq$.
- The merge method m computes the join of two states with respect to $\leq$.
- State is *inflationary* across updates: if u updates a state s to s', then $s \leq s'$.

Shapiro *et al*. show that a state-based object that meets the criteria for a CvRDT is SEC [13].

Differences between CvRDTs and LVars While CvRDTs have much in common with LVars, they differ in the following ways:

- In the CvRDT model, there is no notion of threshold reads; the query operation q reads the exact contents of its local replica, and therefore different replicas may see different states at the same time, if not all updates have been propagated yet. That is, it is possible to observe intermediate states of a CvRDT replica. Such intermediate observations are not possible with LVars.
- In the LVars model, there is no "update" operation that is distinct from "merge"—since LVar `puts` compute the join of the old and new values, *every* LVar update is the analogue of a CvRDT merge operation.
- In the LVars model, we do not have to contend with replication! The LVars model is a shared-memory model, and when an LVar is updated, all reading threads can immediately see the update. In the CvRDT model, updates to a replica are only propagated to other replicas by means of subsequent merge operations.

dates have been delivered to all correct replicas is sufficient to ensure that those replicas have equivalent state, whereas in an object that is merely convergent, there might be some further delay before all replicas agree.

[6] There is a potentially misleading terminology overlap here: the definitions of convergence and strong convergence above pertain not only to CvRDTs (where the C stands for "Convergent"), but to *all* CRDTs.

Each of these differences suggests possibilities for extending the LVars and CvRDT models, bringing them closer together. In this paper, we consider the first two, and propose the following changes:

- Extend the definition of CvRDTs to add a mechanism for specifying LVar-style threshold queries by adding a new g (for "get") operation. Threshold queries made via g would guarantee that the order in which information is added to a CvRDT cannot be observed, ensuring a greater degree of consistency at the price of read availability. A *threshold consistency* property should hold of threshold queries: if they return, they will always return the same value. For a CvRDT where *all* queries are made via g, it will be impossible to observe the order in which updates occur.
- Extend the LVars model to allow non-join update operations—that is, allow update operations other than `put`—while nevertheless preserving determinism. We propose extending the LVar-based calculus of our previous work [9] with a non-join update operation `bump`, discussed below in Section 5. A `bump` operation is crucial for certain applications of LVars, and in fact is already included in a forthcoming release of the LVish library [10], but has not yet been formalized or proved deterministic. We conjecture that support for `bump` will make the LVars model more expressive, while retaining determinism.

4. Bringing Threshold Queries to CvRDTs

In this section, we extend Shapiro *et al*.'s CvRDTs to add support for threshold queries, and we state the threshold consistency property that we claim should hold for CvRDTs extended thusly.

Motivation Since the purpose of CvRDTs is eventual consistency, one might wonder why we would want threshold queries of CvRDTs. After all, if strong consistency were important to us, would we be using CvRDTs in the first place? We argue that the advantage of bringing threshold queries to CvRDTs is that they make it possible to use a single framework—that of lattice-based data structures—for formally reasoning about both strong consistency and eventual consistency. Such a framework is useful because real distributed storage services, such as Amazon's SimpleDB [15], are "multi-consistency": they allow client applications to choose between strong and eventual consistency at the level of individual read operations. The recently proposed Pileus distributed storage system [14] goes a step further and allows consistency choices to be made dynamically, in response to changing network and server load conditions. Such systems motivate the need for tools for formally reasoning about a range of consistency options.

Objects with threshold queries Definition 1 extends Shapiro *et al*.'s definition of a state-based object to add the threshold query method g:

Definition 1 (state-based object with threshold queries)**.** A *state-based object with threshold queries* (henceforth *object*) is a tuple (S, s^0, q, g, u, m), where S is a set of states, $s^0 \in S$ is the initial state, q is a *query method*, g (short for "get") is a *threshold query method*, u is an *update method*, and m is a *merge method*.

We assume a finite set of n processes $p_1, \ldots, p_n$, and consider a single replicated object with one replica at each process, with replica i at process p_i. Processes may crash silently; we say that a non-crashed process is *correct*.

Every replica has initial state s^0. Methods execute at individual replicas, possibly updating that replica's state. The kth method execution at replica i is written $f_i^k(a)$, where f is either q, g, u, or m, and a is the arguments to f. The state of replica i after the kth method execution at i is s_i^k. We say that states s and s' are equivalent, written $s \equiv s'$, if $q(s) = q(s')$.

Next, we can give the definition of a CvRDT supporting threshold queries:

Definition 2 (CvRDT with threshold queries)**.** A *convergent replicated data type with threshold queries* (henceforth *threshold CvRDT*) is an object equipped with a partial order $\leq$, written $(S, \leq, s^0, q, g, u, m)$, that has the following properties:

- S forms a join-semilattice ordered by $\leq$.
- S has a greatest element $\top$ according to $\leq$.
- The merge method m computes the join of two states with respect to $\leq$.
- State is *inflationary* across updates: if u updates a state s to s', then $s \leq s'$.
- The threshold query method g takes a *threshold set* subset S' of S as its argument, blocks at replica i until $s \leq s_i^k$ for some (unique) $s \in S'$ and some k, then unblocks and returns s.

As with LVar threshold reads, the element of a threshold set for which g unblocks is unique because the threshold set is *pairwise incompatible* with respect to $\leq$: every two distinct elements in S' have a join of $\top$.

Causal histories An object's *causal history* is a record of all the updates that have happened at all replicas. The causal history does not track the order in which updates happened, merely that they did happen. The *causal history at replica i after execution k* is the set of all updates that have happened at replica i after execution k. Definition 3 updates Shapiro *et al.*'s definition of causal history for a state-based object to account for g (a trivial change, since execution of g does not change a replica's causal history):

Definition 3 (causal history)**.** A *causal history* is a sequence $[c_1, \ldots, c_n]$, where c_i is a set of the updates that have occurred at replica i. Each c_i is initially $\emptyset$. If the kth method execution at replica i is:

- a query q or a threshold query g, then the causal history at replica i after execution k does not change: $c_i^k = c_i^{k-1}$.
- an update $u_i^k(a)$: then the causal history at replica i after execution k is $c_i^k = c_i^{k-1} \cup u_i^k(a)$.
- a merge $m_i^k(s_{i'}^{k'})$: then the causal history at replica i after execution k is the union of the local and remote histories: $c_i^k = c_i^{k-1} \cup c_{i'}^{k'}$.

We say that an update is *delivered at replica i* if it is in the causal history at replica i.

Eventual consistency and strong eventual consistency With the previous definitions in place, we can define eventual consistency and strong eventual consistency. Informally, eventual consistency means that correct replicas eventually reach the same final value if updates stop arriving. (A *correct replica* is a replica at a correct process.) Formally:

Definition 4 (eventual consistency (EC))**.** An object is *eventually consistent* (EC) if the following three conditions hold (the symbol $\lozenge$ means "eventually"):

- *Eventual delivery*: An update delivered at some correct replica is eventually delivered to all correct replicas:
$$\forall i, j : f \in c_i \implies \lozenge f \in c_j.$$
- *Convergence*: Correct replicas at which the same updates have been delivered eventually have equivalent state:
$$\forall i, j : c_i = c_j \implies \lozenge s_i \equiv s_j.$$
- *Termination*: All method executions terminate.

Finally, we can define strong eventual consistency:

Definition 5 (strong eventual consistency (SEC))**.** An object is *strongly eventually consistent* (SEC) if it is eventually consistent and the following condition holds:

- *Strong convergence*: Correct replicas at which the same updates have been delivered have equivalent state:
$$\forall i, j : c_i = c_j \implies s_i \equiv s_j.$$

It is easy to show that a threshold CvRDT is SEC:

Theorem 1 (strong eventual consistency of threshold CvRDTs)**.** *An object that meets the criteria for a threshold CvRDT is SEC.*

Proof. From Shapiro *et al.*, we have that an object that meets the criteria for a CvRDT is SEC [13]. Since threshold queries do not affect causal history, a threshold CvRDT is SEC as well. □

Threshold consistency Neither eventual consistency nor strong eventual consistency imply that *intermediate* results of the same query q on different replicas of a threshold CvRDT will be consistent. For consistent intermediate results, we must use the threshold query method g, for which we can define a *threshold consistency* property. Threshold consistency does *not* require that the same updates have been delivered to the replicas in question. Instead, if replica i has received a subset of the updates that replica j has received and $g_j(a)$ blocks, then $g_i(a)$ also blocks; but if $g_j(a)$ returns a value s, then $g_i(a)$ may either block or also return s.

Definition 6 (threshold consistency)**.** An object is *threshold consistent* (TC) if, for all i, j:

- If $c_i \subseteq c_j$ and $g_j(a)$ blocks, then $g_i(a)$ blocks.
- If $c_i \subseteq c_j$ and $g_j(a) = s$, then $g_i(a)$ either blocks or returns s.

Claim 1 (threshold consistency of threshold CvRDTs)**.** An object that meets the criteria for a threshold CvRDT is TC.

5. Bringing Inflationary Updates to LVars

As we have seen, CvRDTs support inflationary updates—that is, updates that are nondecreasing with respect to the lattice in question—that are not necessarily join operations. Indeed, the *only* requirement on a CvRDT update is that it be inflationary. With LVars, on the other hand, all updates are joins. In this section we propose adding inflationary non-join updates to the LVars model.

As a canonical example of an LVar update that cannot be directly modeled as a join, consider an atomically incremented counter that occupies one memory location. Atomic increments to such a counter are efficient, commutative, and ultimately fit well inside the LVar framework. Atomic increment is a useful primitive—an example use case is *PhyBin*[7], a bioinformatics application that we parallelized with LVish [10]. PhyBin uses atomic increment operations to gradually build a distance matrix, concurrently incrementing the distance in each cell of the matrix.

More generally, for an LVar with lattice $(D, \leq, \bot, \top)$, a data structure author may define a set of *bump* operations $\texttt{bump}_i : D \to D$, which must meet the following two conditions:

- $\forall a, i.\ \ a \leq \texttt{bump}_i(a)$
- $\forall a, i, j.\ \ \texttt{bump}_i(\texttt{bump}_j(a)) = \texttt{bump}_j(\texttt{bump}_i(a))$

When extending LVish to include `bump`, we must keep in mind that `bump` and `put` operations on the same LVar do not necessarily mix. For example, consider a set of `bump` operations $\{\texttt{bump}_{(+1)}, \texttt{bump}_{(+2)}, \ldots\}$ for atomically incrementing a counter represented by a natural number LVar, with a lattice ordered by the

[7] `http://hackage.haskell.org/package/phybin`

usual $\leq$ on natural numbers. A `put` of 4 and a $\texttt{bump}_{(+1)}$ do not commute! If we start with an initial state of 0 and the `put` occurs first, then the state of the LVar changes to 4 since $\max(0, 4) = 4$, and the subsequent $\texttt{bump}_{(+1)}$ updates it to 5. But if the $\texttt{bump}_{(+1)}$ happens first, then the final state of the LVar will be $\max(1, 4) = 4$. Furthermore, multiple distinct families of `bump` functions only commute among themselves and cannot be combined. In fact, `put` operations themselves, with their underlying join operations on the single state replica, meet the above two conditions for `bump`, and therefore can be viewed as a special case of `bump` that, in addition to being inflationary, also happens to compute a join.

Fortunately, in the LVish library, the author of a particular LVar data structure can choose what operations that data structure should provide, and the Haskell type system will statically rule out programs that attempt to use other, incompatible `bump` operations on the same LVar. Moreover, it is safe to *compose* LVars that support different families of `bump` operations. For example, if we defined a `Counter` LVar type with a `bump` operation, an LVar of type `Set Counter` could contain a monotonically growing set of `bump`-able counters that each monotonically increase. What remains is to formally define the operational semantics of `bump` and update the determinism proofs for LVar calculi [8, 9] to account for it; we expect this to be a straightforward change to the existing proofs.

6. Related Work

Concurrent Revisions The Concurrent Revisions (CR) [11] programming model uses isolation types to distinguish regions of the heap shared by multiple mutators. Rather than enforcing exclusive access, CR clones a copy of the state for each mutator, using a deterministic "merge function" for resolving conflicts at join points.

In CR, variables can be annotated as being shared between a "joiner" thread and a "joinee" thread. Unlike the least-upper-bound writes of LVars, CR merge functions are *not* necessarily commutative; indeed, the default CR merge function is "joiner wins". The programming model allows the programmer to specify which of two writing threads should prevail, regardless of the order in which their writes arrive. The states that a shared variable can take on need not form a lattice. Still, semilattices turn up in the metatheory of CR: in particular, Burckhardt and Leijen [4] show that, for any two vertices in a CR revision diagram, there exists a *greatest common ancestor* state that can be used to determine what changes each side has made—an interesting duality with LVars.

Although versioned variables in CR could model lattice-based data structures—if they used least upper bound as their merge function for conflicts—the programming model nevertheless differs from the LVars model in that effects only become visible at the end of parallel regions, as opposed to the asynchronous communication within parallel regions that the LVars model allows. This semantics precludes the use of traditional lock-free data structures for representing versioned variables.

Bloom and Bloom^L The Bloom language for distributed database programming guarantees eventual consistency for distributed data collections that are updated monotonically. The initial formulation of Bloom [1] had a notion of monotonicity based on set inclusion. More recently, Conway *et al*. [5] generalized Bloom to a more flexible lattice-parameterized system, Bloom^L. Bloom^L combines ideas from CRDTs with *monotonic logic*, resulting in a lattice-parameterized, confluent language that is a close relative of LVish. A monotonicity analysis pass rules out programs that would perform non-monotonic operations on distributed data collections, whereas in the LVars model, monotonicity is enforced by the API presented by LVars. Moreover, since LVish is implemented as a Haskell library (whereas Bloom(^{L}) is implemented as a domain-specific language embedded in Ruby), we can rely on Haskell's static type system for fine-grained effect tracking and monadic encapsulation of LVar effects.

7. Conclusion and Future Work

Although the CvRDT and LVar models were developed independently and serve different purposes, they both leverage lattice properties to ensure that a property of the model holds—determinism in the case of LVars; eventual consistency in the case of CvRDTs. We propose extending CvRDTs to support threshold reads, allowing for reasoning about strong consistency, and extending LVars to support inflationary non-least-upper-bound updates, making a wider range of LVar applications possible while still ensuring determinism. In future work, we will prove the claims that threshold-queryable CvRDTs enforce threshold consistency and that inflationary non-join LVar updates retain determinism.

Acknowledgments

Thanks to the anonymous reviewers, Sam Elliott, and Christopher Meiklejohn for their helpful comments on earlier versions of this paper. This research is supported by NSF grant CCF-1218375.

References

[1] P. Alvaro, N. Conway, J. Hellerstein, and W. R. Marczak. Consistency analysis in Bloom: a CALM and collected approach. In *CIDR*, 2011.

[2] Arvind, R. S. Nikhil, and K. K. Pingali. I-structures: data structures for parallel computing. *ACM Trans. Program. Lang. Syst.*, 11(4), Oct. 1989.

[3] E. Brewer. CAP twelve years later: How the "rules" have changed. `http://www.infoq.com/articles/cap-twelve-years-later-how-the-rules-have-changed`, 2012.

[4] S. Burckhardt and D. Leijen. Semantics of concurrent revisions. In *ESOP*, 2011.

[5] N. Conway, W. Marczak, P. Alvaro, J. M. Hellerstein, and D. Maier. Logic and lattices for distributed programming. In *SOCC*, 2012.

[6] 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 *SOSP*, 2007.

[7] S. Gilbert and N. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2), June 2002.

[8] L. Kuper and R. R. Newton. LVars: lattice-based data structures for deterministic parallelism. In *FHPC*, 2013.

[9] L. Kuper, A. Turon, N. R. Krishnaswami, and R. R. Newton. Freeze after writing: Quasi-deterministic parallel programming with LVars. In *POPL*, 2014.

[10] L. Kuper, A. Todd, S. Tobin-Hochstadt, and R. R. Newton. Taming the parallel effect zoo: Extensible deterministic parallelism with LVish. In *PLDI*, 2014 (to appear).

[11] D. Leijen, M. Fahndrich, and S. Burckhardt. Prettier concurrency: purely functional concurrent revisions. In *Haskell*, 2011.

[12] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. A comprehensive study of Convergent and Commutative Replicated Data Types. Technical Report RR-7506, INRIA, Jan. 2011.

[13] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. Conflict-free replicated data types. In *SSS*, 2011.

[14] D. B. Terry, V. Prabhakaran, R. Kotla, M. Balakrishnan, M. K. Aguilera, and H. Abu-Libdeh. Consistency-based service level agreements for cloud storage. In *SOSP*, 2013.

[15] W. Vogels. Choosing consistency. `http://www.allthingsdistributed.com/2010/02/strong_consistency_simpledb.html`, 2010.

[16] W. Vogels. Eventually consistent. *Commun. ACM*, 52(1), Jan. 2009.