Thesis Proposal:
Lattice-based Data Structures for
Deterministic Parallel and Distributed Programming

Lindsey Kuper
Indiana University

December 6, 2013

Abstract

Deterministic-by-construction parallel programming models guarantee that programs written using them will have the same observable behavior on every run. These models offer the promise of freedom from subtle, hard-to-reproduce bugs caused by schedule nondeterminism. In order to guarantee determinism, though, deterministic-by-construction models must sharply restrict the sharing of state between parallel tasks. Shared state, if it is allowed at all, is typically restricted to a single type of shared data structure, such as single-assignment locations or blocking FIFO queues. These approaches limit the kinds of deterministic algorithms that can be expressed—efficiently or at all—within the model.

This thesis will show that lattice-based data structures, or LVars, are the foundation for a model of deterministic-by-construction parallel programming that allows a more general form of communication between tasks than previously existing guaranteed-deterministic models allowed. LVars generalize existing single-assignment models to allow multiple assignments that are monotonically increasing with respect to an application-specific lattice. They ensure determinism by allowing only monotonic writes and “threshold” reads that block until a lower bound is reached, preventing the order of writes from being observed. After presenting the basic LVars model and showing that it is deterministic, I will show how to extend it in two ways: first, to allow non-blocking “freezing” reads, resulting in a quasi-deterministic model in which programs are guaranteed to behave deterministically modulo exceptions, and second, to allow event handlers, which make it easier to express fixpoint computations with LVars, especially in conjunction with freezing.

Next, I will investigate the relationship between LVars and conflict-free replicated data types (CRDTs), which provide a framework for specifying the behavior of eventually consistent replicated objects in a distributed system. First, I will show how LVar-style threshold reads apply to the setting of CRDTs by extending the definition of state-based CRDTs to allow threshold reads. Threshold reads will guarantee that the order in which information is added to a CRDT cannot be observed, ensuring a greater degree of consistency at the price of read availability. Second, I will use techniques from the CRDT literature to implement LVars that emulate non-monotonic data structures (i.e., counters that support decrement as well as increment and sets that support removal of elements).

Finally, I will demonstrate the viability of the LVars model with LVish, a Haskell library based on it. The LVish library provides a collection of lattice-based data structures, a work-stealing scheduler that runs user-created threads, and a monad in which LVar computations run. LVish leverages Haskell’s type system to index such computations with an effect level to ensure that only certain LVar effects can occur in a given computation, hence statically enforcing determinism or quasi-determinism. I will illustrate the LVish library with running examples and present three case studies that demonstrate its applicability.

1 Introduction

Parallel programming is notoriously difficult. A fundamental reason for this difficulty is that programs can yield inconsistent results, or even crash, due to unpredictable interactions between parallel tasks. Deterministic-by-construction parallel programming models, though, offer the promise of freedom from subtle, hard-to-reproduce nondeterministic bugs in parallel code.
A deterministic-by-construction programming model is one that ensures that all programs written using the model have the same observable behavior every time they are run. How do we define what is observable about a program’s behavior? Certainly, we do not wish to preserve behaviors such as running time across multiple runs—ideally, a deterministic-by-construction parallel program will run faster when more parallel resources are available! Moreover, we do not count the behavior of the scheduler as observable. Indeed, we want to specifically allow tasks to be scheduled dynamically and unpredictably, but without allowing such schedule nondeterminism to affect the outcome of a program. Therefore, in this proposal we define the observable behavior of a program to be the value to which the program evaluates.

1.1 Existing approaches to determinism by construction

Shared state between computations raises the possibility of race conditions that allow schedule nondeterminism to be observed. For instance, if one thread writes 3 to a shared location while another writes 4, then a later thread that reads and returns the location’s contents will nondeterministically return 3 or 4 depending on the order in which the first two threads are scheduled to run. Therefore, deterministic parallel programming models necessarily limit sharing of mutable state between parallel tasks.

One approach is to allow no shared mutable state between tasks, forcing concurrent tasks to produce values independently. An example of no-shared-state parallelism is functional programming with function-level task parallelism, or futures—for instance, in Haskell programs that use the \texttt{par} and \texttt{pseq} combinators [18]. Another approach is pure data parallelism, as in Data Parallel Haskell [25], and yet another is to ensure that the state accessed by concurrent threads is disjoint, as in Deterministic Parallel Java [5].

However, some algorithms are more naturally or efficiently written using shared state or message passing, and hence the development of parallel programming models that allow limited sharing of mutable state while still preserving determinism. Consider two classic deterministic parallel programming models, dating back to the late 1960s and early 1970s:

- In Kahn process networks (KPNs) [15], as well as in the more restricted synchronous data flow systems [16], a network of independent, sequential “computing stations” communicate with each other through blocking FIFO channels. Each station computes a monotonic function from the history of its input channels (i.e., the input it has received so far) to the history of its output channels (the output it has produced so far). KPNs are the basis for deterministic stream-processing languages such as StreamIt [14].

- In parallel single-assignment languages, “full/empty” bits are associated with memory locations so that they may be written to at most once. Single-assignment locations with blocking read semantics are known as IVars [2] and are a well-established mechanism for enforcing determinism in parallel settings: they have appeared in Concurrent ML as SyncVars [27]; in the Intel Concurrent Collections (CnC) system [7]; and have even been implemented in hardware in Cray MTA machines [3]. Although most of these uses incorporate IVars into already-nondeterministic programming environments, the \texttt{monad-par} Haskell library [19] uses IVars in a deterministic-by-construction setting, allowing user-created threads to communicate through IVars without requiring the \texttt{IO} monad. Rather, operations that read and write IVars must run inside a \texttt{Par} monad, thus encapsulating them inside otherwise pure programs, and hence a program in which the only effects are \texttt{Par} effects is guaranteed to be deterministic.

In KPNs and other data-flow models, communication takes place over FIFOs with ever-increasing channel histories, while in IVar-based programming models such as CnC and monad-par, a shared data store of single-assignment memory locations grows monotonically. Hence monotonic data structures—data structures to which information is only added and never removed—are a common theme of guaranteed-deterministic programming models. Yet such programming models emerge independently, without recognition of their common basis. Moreover, they lack generality, since in each case, communication is only permitted through a single type of shared data structure—FIFO queues in KPNs, for instance, or tables of write-once locations in CnC—limiting the kinds of algorithms that can be expressed—efficiently or at all—in the model.

\footnote{We assume that programs have no side effects other than state.}
1.2 LVars: lattice-based monotonic data structures

In this thesis, I will show that lattice-based data structures, or LVars, are the foundation for a model of deterministic-by-construction parallel programming that allows a more general form of communication between tasks than previously existing guaranteed-deterministic models allowed. LVars generalize IVars and are so named because the states 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 (which I will explain in detail in Section 3):

- The put operation can only change an LVar’s state in a way that is monotonically increasing with respect to the user-specified lattice, because it takes the least upper bound of the current state and the new state.
- The get operation allows only limited observations of the state of an LVar. It requires the user to specify 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 least upper bound. A call to get blocks until the LVar in question reaches a (unique) value in the threshold set, then unblocks and returns that value.

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 and get on LVars are the only side effects will have the same observable result in spite of parallel execution and schedule nondeterminism.

1.3 Quasi-deterministic programming with LVars

The LVars model described above guarantees determinism and supports an unlimited variety of shared data structures: anything viewable as a lattice. However, it is not as general-purpose as one might hope. Consider, for instance, an algorithm for unordered graph traversal. A typical implementation involves a monotonically growing set of “seen nodes”; neighbors of seen nodes are fed back into the set until it reaches a fixed point. Such fixpoint computations are ubiquitous, and would seem to be a perfect match for the LVars model due to their use of monotonicity. But they are not expressible using the threshold get and least-upper-bound put operations described above.

The problem is that these computations rely on negative information about a monotonic data structure, i.e., on the absence of certain writes to the data structure. In a graph traversal, for example, neighboring nodes should only be explored if the current node is not yet in the set; a fixpoint is reached only if no new neighbors are found; and, of course, at the end of the computation it must be possible to learn exactly which nodes were reachable (which entails learning that certain nodes were not). I will describe two extensions to the LVars model (again deferring details to Section 3) that make such computations possible:

- First, I will describe how to add a primitive freeze for freezing an LVar, which allows its contents to be read immediately and exactly, rather than the blocking threshold read that get allows. The freeze primitive imposes the following trade-off: once an LVar has been frozen, any further writes that would change its value instead raise an exception; on the other hand, it becomes possible to discover the exact value of the LVar, learning both positive and negative information about it, without blocking. Therefore, LVar programs that use freeze are not guaranteed to be deterministic, because they could nondeterministically raise an exception depending on how put and freeze operations are scheduled. However, we can guarantee that such programs satisfy quasi-determinism: all executions that produce a final value produce the same final value.

- Second, I will describe how to add the ability to attach event handlers to an LVar. When an event handler has been registered with an LVar, it invokes a callback function to run, asynchronously, whenever events arrive (in the form of monotonic updates to the LVar). Crucially, it is possible to check for quiescence of a group of handlers, discovering that no callbacks are currently enabled—a transient, negative property. Since quiescence means that there are no further changes to respond to, it can be used to tell that a fixpoint has been reached.

Of course, since more events could arrive later, there is no way to guarantee that quiescence is permanent—but since the contents of the LVar being written to can only be read through get or freeze operations anyway, early quiescence poses no risk to determinism or quasi-determinism, respectively. In fact, freezing and quiescence work particularly

---

2 As I will explain in Section 3, what I call a “lattice” really need only be a bounded join-semilattice augmented with a greatest element \( \top \).
well together because freezing provides a mechanism by which we can safely “place a bet” that all writes have completed. Hence freezing and handlers make possible fixpoint computations like the graph traversal described above. Moreover, if we can ensure that the freeze does indeed happen after all writes have completed, then we can ensure that the computation is deterministic, and it is possible to enforce this “freeze-last” idiom at the implementation level; see Section 1.5.

1.4 LVars and conflict-free replicated data types

The LVars model I’ve described is closely related to the concept of conflict-free replicated data types (CRDTs) [29] for specifying the behavior of eventually consistent [32] replicated objects in a distributed system. In particular, state-based or convergent replicated data types, abbreviated as CvRDTs, leverage the mathematical properties of join-semilattices to guarantee that all replicas of an object (for instance, in a distributed database) eventually agree. Unlike LVars, CvRDTs allow intermediate states to be observed: if two replicas are updated independently, reads of those replicas may disagree until a (least-upper-bound) merge operation takes place.

Since CvRDTs leverage lattice properties to ensure eventual consistency in the same way that LVars do so to ensure determinism, a sensible next research question is: can we use lessons from the growing literature on replicated data types [29, 28, 8] to improve the LVars model, and vice versa? I will approach this question in both directions:

- First, I will show 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 will prevent intermediate states from being observed, ensuring a greater degree of consistency at the price of read availability.
- Second, I will show how to use techniques from the CRDT literature to develop LVars that support non-monotonic updates: PN-Counters, which can be decremented as well as incremented, and 2P-Sets, from which elements can be removed as well as added.

In Section 4 I give additional background on eventual consistency and CRDTs and discuss further details of how I plan to approach these tasks.

1.5 The LVish library

To demonstrate the practicality of the LVars programming model, I will describe LVish [3], a Haskell library for deterministic and quasi-deterministic programming with LVars. We describe the implementation of LVish in Kuper et al. [23]; rather than covering implementation internals in detail, I will focus on describing the LVish API through examples and case studies.

LVish provides a Par monad for encapsulating parallel computation, and enables a notion of lightweight, library-level threads to be employed with a custom work-stealing scheduler [4]. LVar computations run inside the Par monad, which is indexed by an effect level, allowing fine-grained specification of the effects that a given computation is allowed to perform. For instance, since freeze introduces quasi-determinism, a computation indexed with a deterministic effect level is not allowed to use freeze. Thus, the static type of an LVish computation reflects its determinism or quasi-determinism guarantee. Furthermore, if a freeze is guaranteed to be the last effect that occurs in a computation, then it is impossible for that freeze to race with a put, ruling out the possibility of a run-time put-after-freeze exception. LVish exposes a runParThenFreeze operation that captures this “freeze-last” idiom and has a deterministic effect level.

LVish also provides a variety of lattice-based data structures (e.g., sets, maps, graphs) that support concurrent insertion, but not deletion, during Par computations. In addition to those that LVish provides, users may implement their own lattice-based data structures, and LVish provides tools to facilitate the definition of user-defined LVars. I will describe the proof obligations for data structure implementors and give examples of applications that use user-defined LVars as well as those that the library provides.

---

3 Available at: http://hackage.haskell.org/package/lvish

4 The Par monad exposed by LVish generalizes the original Par monad exposed by the monad-par library (http://hackage.haskell.org/package/monad-par), which allows determinism-preserving communication between threads, but only through IVars, rather than LVars.
I will illustrate LVish through three case studies, drawn from my collaborators’ and my experience using the LVish library, all of which will make use of handlers and freezing:

- First, I will describe using LVish to implement a parallel, pipelined, breadth-first graph traversal in which a (possibly expensive) function \( f \) is mapped over each node in a connected component of a graph. The idea is that the set of results of applying \( f \) to nodes become incrementally available to other computations. For this case study, I will reimplement the breadth-first traversal algorithm that appears in Kuper and Newton [21], which I will update in two ways. First, the original version uses \( \text{runParIO} \) and a “consume” operation that was a precursor to \( \text{freeze} \), and I will refactor it to use the safer \( \text{runParThenFreeze} \). Second, although the original version is pipelined in that it begins invoking \( f \) on already-found nodes before all the nodes in the component have been traversed, it does not take advantage of this early invocation to do something useful with the early result of \( f \); however, it should be straightforward to do so using event handlers.

- Second, I will describe using LVish to parallelize a control flow analysis (\( k \)-CFA) algorithm. The goal of \( k \)-CFA is to compute the flow of values to expressions in a program. The \( k \)-CFA algorithm proceeds in two phases: first, it explores a graph of abstract states of the program; then, it summarizes the results of the first phase. Using LVish, these two phases can be pipelined in a manner similar to the pipelined breadth-first graph traversal described above; moreover, the original graph exploration phase can be internally parallelized. I will contrast the LVish implementation with the original sequential implementation and give performance results.

- Third, I will describe using LVish to parallelize PhyBin [24], a bioinformatics application for comparing genealogical histories (phylogenetic trees) that relies heavily on a parallel tree-edit distance algorithm [30]. In addition to handlers and freezing, the PhyBin application relies on the ability to perform writes to LVars that are commutative and inflationary with respect to the lattice in question, but not idempotent (in contrast to the least-upper-bound writes discussed above, which are idempotent). I will show that these non-idempotent writes, which we call bump operations, preserve determinism as long as programs do not use \( \text{put} \) and \( \text{bump} \) on the same LVar, a property that can be statically enforced by the aforementioned effect specification system in LVish.

## 2 Thesis statement

With the above background, I can state my thesis:

Lattice-based data structures are a general and practical foundation for deterministic and quasi-deterministic parallel and distributed programming.

My dissertation will defend this thesis as follows:

- **Lattice-based data structures**: I will formally define LVars and use them to define \( \lambda_{LVish} \), a call-by-value parallel calculus with shared state, including a runnable version implemented in PLT Redex [12] for interactive experimentation.

- **general**: I will defend the generality of the LVars model by showing how previously existing deterministic parallel programming models can be expressed with LVish. This is possible because the definition of \( \lambda_{LVish} \) is parameterized by the choice of lattice. For example, a lattice of channel histories (phylogenetic trees) that relies heavily on a parallel tree-edit distance algorithm [30]. In addition to handlers and freezing, the PhyBin application relies on the ability to perform writes to LVars that are commutative and inflationary with respect to the lattice in question, but not idempotent (in contrast to the least-upper-bound writes discussed above, which are idempotent). I will show that these non-idempotent writes, which we call bump operations, preserve determinism as long as programs do not use \( \text{put} \) and \( \text{bump} \) on the same LVar, a property that can be statically enforced by the aforementioned effect specification system in LVish.

- **practical**: I will defend the practicality of LVars by describing the LVish Haskell library and demonstrating how it is used for practical programming with the three case studies described above, including performance results.

- **deterministic**: I will defend the claim that the basic LVars model guarantees determinism by giving a proof of determinism for \( \lambda_{LVish} \), including the aforementioned \( \text{put} \), \( \text{get} \), and non-idempotent bump operations on LVars.

\(^5\) \( \lambda_{LVish} \) is the LVish calculus described in Kuper et al. [23]. I call it \( \lambda_{LVish} \) here to avoid confusion with the LVish library discussed in Section 1.5.
• **quasi-deterministic**: I will defend this claim by giving a proof of quasi-determinism for the full λLVish calculus with the additions of the `freeze` operation and event handlers.

• **distributed programming**: I will defend the claim that LVars are applicable to distributed programming by showing how LVar-style threshold reads apply to the setting of CRDTs, and by showing how techniques for implementing non-monotonic CRDTs can be used to implement LVars that support non-monotonic operations as well.

3 Technical overview

In this section I summarize the key technical aspects of the λLVish calculus and its quasi-determinism proof. λLVish is a quasi-deterministic, parallel, call-by-value λ-calculus extended with a store containing LVars. Its definition is parameterized by an application-specific lattice, representing the set of states that LVars in the calculus can take on and the ordering in which they can be taken on.

For brevity, I have omitted some details from this section; complete details can be found in our previously published work [21, 20, 23, 22].

3.1 Lattices

The application-specific lattice is given as a 4-tuple $(D, \sqsubseteq, \bot, \top)$ where $D$ is a set, $\sqsubseteq$ is a partial order on the elements of $D$, $\bot$ is the least element of $D$ according to $\sqsubseteq$, and $\top$ is the greatest. The $\bot$ element represents the initial “empty” state of every LVar, while $\top$ represents the “error” state that would result from conflicting updates to an LVar. The partial order $\sqsubseteq$ represents the order in which an LVar may take on states. It induces a binary least upper bound (lub) operation $\sqcup$ on the elements of $D$. We require that every two elements of $D$ have a least upper bound in $D$. Intuitively, the existence of a lub for every two elements of $D$ means that it is possible for two subcomputations to independently update an LVar, and then deterministically merge the results by taking the lub of the resulting two states. Formally, this makes $(D, \sqsubseteq, \bot, \top)$ a bounded join-semilattice with a designated greatest element ($\top$). For brevity, we use the term “lattice” as shorthand for “bounded join-semilattice with a designated greatest element”.

Figure 1 gives three examples of lattices for common data structures. The simplest example of a useful lattice is one that represents the state space of a single-assignment variable (an IVar). A natural-number-valued IVar, for instance, would correspond to the lattice in Figure 1(b), that is, $(\{\top, \bot\} \cup \mathbb{N}, \sqsubseteq, \bot, \top)$, where $\sqsubseteq$ is defined by setting $\bot \sqsubseteq d \sqsubseteq \top$ and $d \sqsubseteq d$ for all $d \in D$. This is a lattice of height three and infinite width, where the naturals are arranged horizontally. After the initial write of some $n \in \mathbb{N}$, any further conflicting writes would push the state of the IVar to $\top$ (an error). For instance, if one thread writes 2 and another writes 1 to an IVar (in arbitrary order), the second of the two writes would result in an error because $2 \sqcup 1 = \top$.

In the lattice of Figure 1(a), on the other hand, the $\top$ element can never be reached, because the least upper bound of any two writes is just the maximum of the two. For instance, if one thread writes 2 and another writes 1 to an IVar (in arbitrary order), the second of the two writes would result in an error because $2 \sqcup 1 = \bot$.

3.2 Freezing

To model freezing, the representation of an LVar must include information about whether it is “frozen” or not. Thus, in our model an LVar’s state is a pair $(d, frz)$, where $d$ is an element of the application-specific set $D$ and $frz$ is a “status bit” of either true or false. We can define an ordering $\sqsubseteq_p$ on LVar states $(d, frz)$ in terms of the application-specific ordering $\sqsubseteq$ on elements of $D$. Every element of $D$ except $\top$ corresponds to a value at which the LVar is “freezable”.

In practice, different LVars will correspond to different lattices. Multiple lattices can in principle be encoded using a sum construction, so the choice of parameterizing the language by a single lattice is just to keep the presentation simple; in any case, the LVish library implementation supports multiple lattices natively.
Figure 1: Example LVars 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 \( \text{getSnd} \) causes it to unblock and return a result.

- Two unfrozen states are ordered according to the application-specific \( \sqsubseteq \); that is, \((d, \text{false}) \sqsubseteq_p (d', \text{false})\) exactly when \(d \sqsubseteq d'\).
- Two frozen states do not have an order, unless they are equal: \((d, \text{true}) \sqsubseteq_p (d', \text{true})\) exactly when \(d = d'\).
- An unfrozen state \((d, \text{false})\) is less than or equal to a frozen state \((d', \text{true})\) exactly when \(d \sqsubseteq d'\).
- The only situation in which a frozen state is less than an unfrozen state is if the unfrozen state is \(\top\); that is, \((d, \text{true}) \sqsubseteq_p (d', \text{false})\) exactly when \(d' = \top\).

The addition of status bits to the application-specific lattice results in a new lattice \((D_p, \sqsubseteq_p, \bot_p, \top_p)\), and we write \(\sqcup_p\) for the least upper bound operation that \(\sqsubseteq_p\) induces.

### 3.3 Stores

During the evaluation of \(\lambda_{LVish}\) programs, a store \(S\) keeps track of the states of LVars. Each LVar is represented by a binding from a location \(l\), drawn from a set \(\text{Loc}\), to its state, which is some pair \((d, \text{frz})\) from the set \(D_p\).

We use the notation \(S[l \mapsto (d, \text{frz})]\) to denote extending \(S\) with a binding from \(l\) to \((d, \text{frz})\). If \(l \in \text{dom}(S)\), then \(S[l \mapsto (d, \text{frz})]\) denotes an update to the existing binding for \(l\), rather than an extension. We can also denote a store by explicitly writing out all its bindings, using the notation \([l_1 \mapsto (d_1, \text{frz}_1), l_2 \mapsto (d_2, \text{frz}_2), \ldots]\). It is straightforward to lift the \(\sqsubseteq_p\) operations defined on elements of \(D_p\) to a \(\sqsubseteq_S\) operation defined on stores. Stores ordered by \(\sqsubseteq_S\) also form a lattice (with bottom element \(\emptyset\) and top element \(\top_S\)); we write \(\sqcup_S\) for the induced lub operation. If, for example,

\[
(d_1, \text{frz}_1) \sqcup_p (d_2, \text{frz}_2) = \top_p,
\]

then

\[
[l \mapsto (d_1, \text{frz}_1)] \sqcup_S [l \mapsto (d_2, \text{frz}_2)] = \top_S.
\]

A store containing a binding \(l \mapsto (\top, \text{frz})\) can never arise during the execution of an \(\lambda_{LVish}\) program, because, as we will see in Section 3.5, an attempted \texttt{put} that would take the value of \(l\) to \(\top\) will raise an error.
Given a lattice \((D, \subseteq, \bot, \top)\) with elements \(d \in D:\)

\[
\begin{align*}
\text{configurations } \sigma & := (S; e) \mid \text{error} \\
\text{expressions } e & := x \mid v \mid e \mid c \mid \text{get } e \mid \text{put } e \mid \text{new } \mid \text{freeze } e \\
& \quad \mid \text{freeze } e \text{ after } e \text{ with } e \\
& \quad \mid \text{freeze } l \text{ after } Q \text{ with } \lambda x. e, \{e, \ldots\}, H
\end{align*}
\]

\[
\begin{align*}
\text{stores } S & := [l_1 \mapsto p_1, \ldots, l_n \mapsto p_n] \mid \top_S \\
\text{values } v & := () \mid d \mid p \mid l \mid P \mid Q \mid \lambda x. e
\end{align*}
\]

\[
\begin{align*}
\text{eval contexts } E & := [] \mid E e \mid e E \mid \text{get } E e \mid \text{get } e E \mid \text{put } E e \\
& \quad \mid \text{put } e E \mid \text{freeze } E \mid \text{freeze } e \text{ after } e \text{ with } e \\
& \quad \mid \text{freeze } e \text{ after } E \text{ with } e \mid \text{freeze } e \text{ after } e \text{ with } E \\
& \quad \mid \text{freeze } e \text{ after } E \text{ with } e \mid \text{freeze } e \text{ after } e \text{ with } E
\end{align*}
\]

“handled” sets \(H := \{d_1, \ldots, d_n\}\)  
states \(p := (d, \text{frz})\)

\[
\begin{align*}
\text{threshold sets } P & := \{p_1, p_2, \ldots\} \\
\text{event sets } Q & := \{d_1, d_2, \ldots\} \\
\text{status bits } \text{frz} & := \text{true} \mid \text{false}
\end{align*}
\]

Figure 2: Syntax for \(\lambda_{LVish}\).

### 3.4 The \(\lambda_{LVish}\) Language

The syntax and operational semantics of the \(\lambda_{LVish}\) calculus appear in Figures 2 and 3, respectively. As we have noted, both the syntax and semantics are parameterized by the lattice \((D, \subseteq, \bot, \top)\). The reduction relation \(\longrightarrow\) is defined on configurations \((S; e)\) comprising a store and an expression. The error configuration, written \text{error}, is a unique element added to the set of configurations, but we consider \((\top_S; e)\) to be equal to \text{error} for all expressions \(e\). The metavariable \(\sigma\) ranges over configurations.

\(\lambda_{LVish}\) uses a reduction semantics based on evaluation contexts. The \(\text{E-EVAL-CTX}\) rule is a standard context rule, allowing reductions to apply within a context. The choice of context determines where evaluation can occur; in \(\lambda_{LVish}\), the order of evaluation is nondeterministic (that is, a given expression can generally reduce in more than one way), and so it is generally not the case that an expression has a unique decomposition into redex and context. For example, in an application \(e_1 e_2\), either \(e_1\) or \(e_2\) might reduce first. The nondeterminism in choice of evaluation context reflects the nondeterminism of scheduling between concurrent threads, and in \(\lambda_{LVish}\), the arguments to get, put, freeze, and application expressions are implicitly evaluated concurrently.

Arguments must be fully evaluated, however, before function application (\(\beta\)-reduction, modeled by the \(\text{E-BETA}\) rule) can occur. We can leverage this property to define \text{let} par as syntactic sugar:

\[
\text{let } \text{par } x = e_1; y = e_2 \text{ in } e_3 \triangleq ((\lambda x. (\lambda y. e_3)) e_1) e_2
\]

Because we do not reduce under \(\lambda\)-terms, we can sequentially compose \(e_1\) before \(e_2\) by writing \text{let } \_ = e_1 \text{ in } e_2, which desugars to \((\lambda _- e_2) e_1\). Sequential composition is useful, for instance, when allocating a new LVar before beginning a set of side-effecting put/get/freeze operations on it.

### 3.5 Semantics of new, put, and get

In \(\lambda_{LVish}\), the new, put, and get operations respectively create, write to, and read from LVars in the store:

- **new** (implemented by the \(\text{E-NEW}\) rule) extends the store with a binding for a new LVar whose initial state is \((\bot, \text{false})\), and returns the location \(l\) of that LVar (i.e., a pointer to the LVar).

- **put** (implemented by the \(\text{E-PUT}\) and \(\text{E-PUT-ERR}\) rules) takes a pointer to an LVar and a new lattice element \(d_2\) and updates the LVar’s state to the least upper bound of the current state and \((d_2, \text{false})\), potentially pushing the state of the LVar upward in the lattice. Any update that would take the state of an LVar to \(\top_p\) results in the program immediately stepping to \text{error}.
Given a lattice \((D, \sqsubseteq, \bot, \top)\) with elements \(d \in D\):

\[
\text{incomp}(P) \triangleq \forall p_1, p_2 \in P. \ (p_1 \neq p_2 \implies p_1 \sqcup p_2 = \top_p)
\]

**Figure 3:** An operational semantics for \(\lambda_{L\text{Vish}}\).

- get (implemented by the E-Get rule) performs a blocking threshold read. It takes a pointer to an LVar and a threshold set \(P\), which is a non-empty set of LVar states that must be pairwise incompatible, expressed by the premise \(\text{incomp}(P)\). A threshold set \(P\) is pairwise incompatible iff the lub of any two distinct elements in \(P\) is \(\top_p\). If the LVar’s state \(p_1\) in the lattice is at or above some \(p_2 \in P\), the get operation unblocks and returns \(p_2\).

Note that \(p_2\) is a unique element of \(P\), for if there is another \(p_2' \neq p_2\) in the threshold set such that \(p_2' \sqsubseteq p_1\), it would follow that \(p_2' \sqcup p_2 = p_1 \neq \top_p\), which contradicts the requirement that \(P\) be pairwise incompatible.

Is the get operation deterministic? Consider two lattice elements \(p_1\) and \(p_2\) that have no ordering and have \(\top_p\) as their lub, and suppose that puts of \(p_1\) and \(p_2\) and a get with \(\{p_1, p_2\}\) as its threshold set all race for access to an LVar \(lv\). Eventually, the program is guaranteed to raise an exception, because \(p_1 \sqcup p_2 = \top_p\), but in the meantime, get \(lv\) \(\{p_1, p_2\}\) could return either \(p_1\) or \(p_2\). Therefore, get can behave nondeterministically—but this behavior is not observable in the final answer of the program, which is guaranteed to subsequently raise an exception.

As an example of a threshold read, consider an LVar \(lv\) whose states form a lattice of pairs of natural-number-valued IVars; that is, \(lv\) is 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). We can then define get\text{Fst} and get\text{Snd} operations for reading from the first and second entries of \(lv\):

\[
\text{get\text{Fst}} \ p \triangleq \text{get} \ p \ \{m, \bot\} \ | \ m \in \mathbb{N}
\]

\[
\text{get\text{Snd}} \ p \triangleq \text{get} \ p \ \{(\bot, n)\} \ | \ n \in \mathbb{N}
\]

\[\text{Although incomp}(P)\] is given as a premise of the E-Get reduction rule (suggesting that it is checked at runtime), in the LVish library implementation threshold sets are not written explicitly, and it is the data structure author’s responsibility to ensure that any provided read operations have threshold semantics.
This allows us to write programs like the following:

```
let par = put lv (⊥, 4)
  = put lv (3, ⊥)
  x = getSnd lv
in x
```

In the call `getSnd lv`, the threshold set is `{(⊥, 0), (⊥, 1), ...}`, 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 `{(⊥, 0), (⊥, 1), ...}` can ever be reached. The call, `getSnd lv` may unblock and return `(⊥, 4)` any time after the second entry of `lv` has been written, regardless of whether the first entry has been written yet. It is therefore possible to use LVars to safely read parts of an incomplete data structure—say, an object that is in the process of being initialized by a constructor.

3.6 Semantics of freezing and event handling

The $\lambda_{LVish}$ calculus includes a simple form of `freeze` that immediately freezes an LVar (see E-FREEZE-SIMPLE). More interesting is the `freeze - after - with` primitive, which models the use of freezing together with event handling. The expression `freeze e_lv after e_events with e_cb` has the following semantics:

- It attaches the callback `e_cb` to the LVar `e_lv`. The expression `e_events` must evaluate to a event set `Q`; the callback will be executed, once, for each lattice element in `Q` that the LVar’s state reaches or surpasses. The callback `e_cb` is a function that takes a lattice element as its argument. Its return value is ignored, so it runs solely for effect. For instance, a callback might itself do a `put` to the LVar to which it is attached, triggering yet more callbacks.

- If the handler reaches a quiescent state, the LVar `e_lv` is frozen, and its exact state is returned (rather than an underapproximation of the state, as with `get`).

To keep track of the running callbacks, $\lambda_{LVish}$ uses an auxiliary form,

```
freeze l after Q with λx.e_0, {e,...}, H
```

where:

- The value `l` is the LVar being handled/frozen;
- The set `Q` (a subset of the lattice $D$) is the event set;
- The value $\lambda x. e_0$ is the callback function;
- The set of expressions `{e,...}` are the running callbacks; and
- The set $H$ (a subset of the lattice $D$) represents those values in $Q$ for which callbacks have already been launched.

Due to the use of evaluation contexts, any running callback can execute at any time, as if each is running in its own thread. The rule E-SPAWN-HANDLER launches a new callback thread any time the LVar’s current value is at or above some element in $Q$ that has not already been handled. This step can be taken nondeterministically at any time after the relevant `put` has been performed.

The rule E-FREEZE-FINAL detects quiescence by checking that two properties hold. First, every event of interest (lattice element in $Q$) that has occurred (is bounded by the current LVar state) must be handled (be in $H$). Second, all existing callback threads must have terminated with a value. In other words, every enabled callback has completed. When such a quiescent state is detected, E-FREEZE-FINAL freezes the LVar’s state. Like E-SPAWN-HANDLER, the rule can fire at any time, nondeterministically, that the handler appears quiescent—a transient property! But after the
Lemma 1 (Strong Local Quasi-Confluence)

Therefore, freezing is a way of “betting” that once a collection of callbacks have completed, no further puts that change the LVar’s value will occur. For a given run of a program, either all puts to an LVar arrive before it has been frozen, in which case the value returned by freeze − after − with is the lub of those values, or some put arrives after the LVar has been frozen, in which case the program will step to error. And thus we have arrived at quasi-determinism: a program will always either evaluate to the same answer or it will raise an exception.

To ensure that we will win our bet, we need to guarantee that quiescence is a permanent state, rather than a transient one—that is, we need to perform all puts either prior to freeze − after − with, or by the callback function within it (as will be the case for fixpoint computations). In practice, freezing is usually the last step of an algorithm, permitting its result to be extracted. Our implementation provides a special runParThenFreeze function that does so, and thereby guarantees full determinism.

3.7 Quasi-determinism proof outline

The proof of quasi-determinism for λLVish formalizes the claim in Section 1 that, for a given program, although some executions may raise exceptions, all executions that produce a final result will produce the same final result. In this section, I give the statements of the main quasi-determinism theorem and the two most important supporting lemmas. The statements of the remaining lemmas, and proofs of all our theorems and lemmas, are included in our technical report.

The main result, Theorem 1, says that if two executions starting from a configuration σ terminate in configurations σ′ and σ″, then σ′ and σ″ are the same configuration, or one of them is error.

Theorem 1 (Quasi-Determinism). If σ −→ σ′ and σ −→ σ″, and neither σ′ nor σ″ can take a step, then either:

1. σ′ = σ″ up to a permutation on locations π, or
2. σ′ = error or σ″ = error.

Theorem 1 follows from a series of quasi-confluence lemmas. The most important of these, Strong Local Quasi-Confluence (Lemma 1), says that if a configuration steps to two different configurations, then either there exists a single third configuration to which they both step (in at most one step), or one of them steps to error. Additional lemmas generalize Lemma 1’s result to multiple steps by induction on the number of steps, eventually building up to Theorem 1.

Lemma 1 (Strong Local Quasi-Confluence). If σ ≡ (S; e) −→ σa and σ −→ σb, then either:

1. there exist π, i, j and σc such that σa −→i σc and σb −→j π(σc) and i ≤ 1 and j ≤ 1, or
2. σa −→ error or σb −→ error.

In order to show Lemma 1 we need a “frame property” for λLVish that captures the idea that independent effects commute with each other. Lemma 2, the Independence lemma, establishes this property. Consider an expression e that runs starting in store S and steps to e′, updating the store to S′. The Independence lemma allows us to make a double-edged guarantee about what will happen if we run e starting from a larger store S ⊔ S″: first, it will update the store to S′ ⊔ S″; second, it will step to e′ as it did before. Here S ⊔ S″ is the least upper bound of the original S and some other store S″ that is “framed on” to S; intuitively, S″ is the store resulting from some other independently-running computation.

Lemma 2 (Independence). If (S; e) −→ (S′; e′) (where ⟨S′; e′⟩ ≠ error), then we have that:

(S ⊔ S″; e) −→ (S′ ⊔ S″; e′),

where S″ is any store meeting the following conditions:

• S″ is non-conflicting with ⟨S; e⟩ −→ ⟨S′; e′⟩,
• S′ ⊔ S″ ≡fr S, and

11
Lemma 2 requires as a precondition that the stores \( \mathcal{S}' \sqcup \mathcal{S} \) and \( \mathcal{S} \) are equal in status—that, for all the locations shared between them, the status bits of those locations agree. This assumption rules out interference from freezing. Finally, the store \( \mathcal{S}'' \) must be non-conflicting with the original transition from \( \langle \mathcal{S}; e \rangle \) to \( \langle \mathcal{S}'; e' \rangle \), meaning that locations in \( \mathcal{S}'' \) cannot share names with locations newly allocated during the transition; this rules out location name conflicts caused by allocation.

**Definition 1.** Two stores \( \mathcal{S} \) and \( \mathcal{S}' \) are equal in status (written \( \mathcal{S} =_{\text{frz}} \mathcal{S}' \)) iff for all \( l \in (\text{dom}(\mathcal{S}) \cap \text{dom}(\mathcal{S}')) \), if \( \mathcal{S}(l) = (d, \text{frz}) \) and \( \mathcal{S}'(l) = (d', \text{frz}') \), then \( \text{frz} = \text{frz}' \).

**Definition 2.** A store \( \mathcal{S}'' \) is non-conflicting with the transition \( \langle \mathcal{S}; e \rangle \rightarrow \langle \mathcal{S}'; e' \rangle \) iff \( (\text{dom}(\mathcal{S}'') - \text{dom}(\mathcal{S})) \cap \text{dom}(\mathcal{S}'') = \emptyset \).

## 4 Joining forces: LVars and conflict-free replicated data types

In this section, I discuss the relationship between the LVars model I’ve described and the concept of conflict-free replicated data types, and explain how I will leverage this relationship.

### 4.1 Replication 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 [13] of distributed computing describes a three-way trade-off among

- consistency, in which every replica sees the same information;
- availability, in which all information is available for both reading and writing by all replicas; and
- partition tolerance, in which the system is robust to parts of it being unable to communicate with one another.

An (oversimplified, but useful to a first approximation) interpretation of the CAP theorem is the slogan, “Consistency, availability, and partition tolerance: pick at most two.” In practice, real systems must be robust to network partitions and hence must compromise on at least one of consistency or availability. Moreover, though, consistency, availability, and partition tolerance 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 allow less consistency accordingly. Highly available distributed systems (e.g., Dynamo [11]) give up on strict consistency in favor of eventual consistency [32], in which replicas may not always agree, but if updates stop arriving, all replicas will eventually come to agree.

### 4.2 Resolving conflicts between replicas

How can eventually consistent systems ensure that all replicas of an object come to agree? In particular, if replicas differ, how do we determine which is “right”? 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.

As a more practical proposal, we could try to determine which replica was written most recently, then declare the last written replica the winner. But this approach is also less than ideal: even if we had a way of perfectly 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 developers of Dynamo, Amazon’s distributed key-value store, acknowledge this in their discussion of application-specific mechanisms for resolving conflicts between replicas [11]:

- \( \mathcal{S}' \sqcup \mathcal{S} \neq \top \mathcal{S} \).
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 operation.

This notion of application-specific conflict resolution is not without its problems, especially if implemented in an ad-hoc way. Fortunately, we need not implement it in an ad-hoc way: conflict-free replicated data types (CRDTs) give us a simple mathematical framework for reasoning about and enforcing the eventual consistency of replicated objects in a distributed system. In the following sections, I review the basics of CRDTs from the work of Shapiro et al. [29, 28], then discuss the relationship between CRDTs and LVars.

4.3 Eventual consistency and CRDTs

Before discussing CRDTs, we must formally define 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. Along with convergence, they give two other conditions that are required for eventual consistency: eventual delivery, meaning that all replicas receive all update messages, and termination, meaning that all method executions terminate (see below for more on methods).

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\(^8\). A conflict-free replicated data type (CRDT), then, is a data type (i.e., a specification for an object) satisfying certain 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 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. The state-based (CvRDT) style is closer to the LVars model, although, as we will see, CmRDTs can emulate CvRDTs, and vice versa.

4.4 CvRDTs and LVars

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 every replica regularly\(^9\) sends its local state to some other replica, 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 \((S, \leq, s^0, q, u, m)\), that has the following properties:

---

8 Indeed, as noted in the Dynamo article, Amazon’s shopping cart presents an anomaly whereby an item removed from a cart may re-appear!
9 Contrast with ordinary convergence, in which replicas only eventually have equivalent state. An object might be convergent but not strongly convergent if, for example, the system executes an update on some replica and then later rolls back the update after discovering that it conflicts with another.
10 This terminology is somewhat unfortunate: the definitions of convergence and strong convergence above pertain to all CRDTs, not just those specified in the “convergent” style!
11 “Infinitely often”, according to Shapiro et al.
- $S$ forms a join-semilattice ordered by $\leq$.
- The merge method $m$ computes the least upper bound of the 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 [29]. CvRDTs have much in common with LVars, but 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.
- In the LVars model, there is no “update” operation that is distinct from “merge”—since LVar puts compute the least upper bound of the old and new values, an LVar update is the equivalent 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 immediately see the update.

These differences, however, suggest some possibilities for extending both the LVars and CvRDT models. I propose to do the following:

- Extend the definition of CvRDTs to add a mechanism for specifying LVar-style threshold reads—say, by adding a new $g$ (for “get”) operation. Threshold reads should 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. My hypothesis is that, for a CvRDT where all reads are threshold reads made via the $g$ operation, we can guarantee something more than strong convergence: all reads will be consistent. I will formally define and prove this query consistency property for extended CvRDTs. In practice, though, a combination of threshold reads and regular reads may make the most sense.

- Extend the LVars model to allow non-lub update operations—that is, allow update operations other than put—while nevertheless preserving determinism. In fact, the bump operation discussed in Section 1.5—which is already a useful part of the LVish library, but not yet formalized or proved deterministic in the setting of $\lambda_{LVish}$—is just such a non-lub update operation. I will parameterize the definition of $\lambda_{LVish}$ by an inflationary function $f$, add a bump expression to the $\lambda_{LVish}$ syntax, add (something like) the following rule $E$-BUMP to the operational semantics, and prove determinism for the resulting extended language:

  $E$-BUMP
  
  \[
  \frac{S(l) = (d_1, \text{false}) \quad d_1 \sqsubseteq f(d_1) \quad f(d_1) \neq \top}{\langle S; \text{bump } l \rangle \leftrightarrow \langle S[l \mapsto (f(d_1), \text{false})]; () \rangle}
  \]

  Support for inflationary non-lub updates should make the LVars model more expressive (while retaining determinism) and bring the LVars and CvRDT models closer together.

Finally, there is another interesting possibility, which I discuss next: the existence of CRDTs that allow non-monotonic operations suggests a way of extending LVars to allow not only non-lub updates, but non-monotonic updates.

4.5 CmRDTs and non-monotonic updates

As mentioned above, CmRDTs and CvRDTs can emulate each other. However, unlike CvRDTs, which are state-based objects specified using partial orders and for which replicas converge by exchanging information about their local state, CmRDTs are op-based objects, and their replicas converge by telling each other what operations have taken place locally.

Shapiro et al. specify an op-based object as a tuple ($S, s^0, q, t, u, P$), where, as before, $S$, $s^0$, and $q$ are respectively the set of states, the initial state, and the query method. $t$ and $u$ are the prepare-update method and the effect-update method, respectively: updating a local replica involves executing $t$ locally, immediately followed by $u$ locally, and so
we denote an update as a pair \((t, u)\). However, the effect-update method \(u\) also executes at all other replicas. The local replica delivers the effect-update to the other replicas using a communication protocol, specified by the delivery precondition \(P\), that makes certain assumptions about the reliability and consistency of the underlying network.

For present purposes, the details of \(P\) are not important; the important thing to note is that an op-based object has no merge method, and so the above definition of CvRDTs, in which the merge method \(m\) computes a least upper bound, cannot apply to op-based objects. Instead, Shapiro et al. define an op-based or commutative replicated data type (CmRDT) to be an op-based object with the property that all concurrent updates commute. We say that updates \((t, u)\) and \((t', u')\) commute iff, for any reachable replica state \(s\) where both \(u\) and \(u'\) are enabled by the delivery precondition \(P\) (that is, \(P(s, u)\) and \(P(s, u')\) are both true), the following conditions hold:

- \(u\) remains enabled by \(P\) after \(u'\) has executed,
- \(u'\) remains enabled by \(P\) after \(u\) has executed, and
- the state resulting from executing \(u\) followed by \(u'\) starting from \(s\) is equivalent to the state resulting from executing \(u'\) followed by \(u\) starting from \(s\).

As an example of a CmRDT, Shapiro et al. give an op-based specification for an OR-Set (“observed-remove set”) data structure. OR-Sets are sets that support removal as well as addition of elements (because concurrent adds and removes commute).

Just as a state-based object that meets the CvRDT criteria is SEC, Shapiro et al. show that an op-based object that meets the criteria for a CmRDT is SEC [29]. Moreover, they give a constructive proof that CmRDTs and CvRDTs can emulate each other by showing a general way to construct an equivalent CvRDT given a CmRDT, and vice versa. In the CmRDT-to-CvRDT direction of the proof, known updates (operations that have taken place locally) and delivered updates (local operations that have been broadcast by an effect-update) are modeled as two grow-only sets.

Since CmRDT operations do not have to update the state monotonically with respect to a lattice, it is possible to implement CmRDTs (and therefore equivalent CvRDTs!) that allow “non-monotonic” operations. (The state of the emulating CvRDT, however, does grow monotonically with respect to its partial order; otherwise, it would not be a CvRDT.) For example, it should be possible to use Shapiro et al.’s emulation construction to emulate their op-based OR-Set specification with an equivalent state-based one.

Shapiro et al. also give examples showing that, for certain data types, it is possible to directly specify a CvRDT that allows non-monotonic operations. A set that allows both additions and removals, for instance, can be specified directly as a CvRDT; the trick is to encode the set as two grow-only sets, one for elements added and one for elements removed. This data structure is known as a 2P-Set (for “two-phase set”). However, with such an encoding, once an element is removed, it can never be added again thereafter! Counters, however, do not suffer from the same problem: we can encode a counter that allows both increments and decrements directly as a CvRDT using two grow-only counters, one for increments and one for decrements, and having the query method \(q\) return the difference of the two. The resulting data structure is called a PN-Counter (for “positive/negative counter”).

I refer to CvRDTs that allow non-monotonic operations as monotonic emulations of non-monotonic data structures, whether they use Shapiro et al.’s CmRDT-to-CvRDT construction or whether they are directly implemented as CvRDTs. However, queries can observe the order of updates—and allowing non-monotonic operations aggravates this observability problem. Next, I consider what it would take to prevent the order of updates from being observed even in the presence of non-monotonic updates.

4.6 Threshold reads of non-monotonic data structures

What would it mean to perform a threshold read on a data structure that allows non-monotonic operations? At first glance, threshold reads of non-monotonic data structures would seem to pose a problem for determinism. If only monotonic operations are allowed, as is the case for LVars, then whether or not a given \(get\) operation unblocks is the same from run to run (although it may unblock at different times, depending on scheduling). But if the state of an LVar were allowed to move both up and down in a lattice, this would no longer be the case. For instance, if a \(get\) operation were thresholded on an element that was added to and later removed from a set, then the \(get\) might block forever, or not, depending on whether it ran after or before the element’s removal.
One option for resolving this dilemma is to give up on threshold reads entirely for data structures that allow non-monotonic operations. However, it is also worth considering another option, which is to change the semantics of threshold reads to reveal even less information than they do now. In the existing LVars model, when a get returns, it is a guarantee that the contents of the LVar it is reading are at or above a certain point in the lattice: “The contents of this LVar are at least $x$.” But for non-monotonic data structures, we could, instead, have the following guarantee: “There exists a schedule under which the contents of this data structure could have at one time been $x$. ” Although this latter guarantee is much weaker, there still may be situations in which it is useful. Changing the semantics of threshold reads in this way would allow thresholding on the state of a CmRDT directly.

Finally, yet another possibility is to leave the semantics of threshold reads as they are and threshold directly on the (monotonically growing) state of a CvRDT that is the monotonic emulation of a non-monotonic data structure. Such a CvRDT might be an emulation of a CmRDT, or it might be a non-monotonic data structure encoded directly as a CvRDT, such as a PN-Counter. Threshold reads of the emulating (monotonically growing) data structure, rather than the emulated (non-monotonic) one, could leverage what we know about the implementation details of the emulation to allow unusual kinds of queries. It might be useful, for instance, to know when a PN-Counter has been decremented at least $n$ times, regardless of its total. If so, we could easily threshold on the size of the internal grow-only decremen counter using a standard threshold read.

The last of these possibilities seems to fit well into the LVars model because it does not require changing the monotonic semantics of LVars. Therefore, I propose to implement LVar-based versions of 2P-sets and PN-Counters, using a monotonic emulation for the non-monotonic operations, and exposing get operations that correspond to standard threshold reads, but threshold on the emulating monotonic data structure.

5 Related work

Work on deterministic parallel programming models is long-standing. As discussed in Section 1, the LVars model builds on long traditions of work on parallel programming models based on monotonically-growing shared data structures, and it provides a framework for generalizing and unifying these existing approaches. In this section I describe some more recent contributions to the literature.

As we have seen, what deterministic parallel programming models have in common is that they all must do something to restrict access to mutable state shared among concurrent computations so that schedule nondeterminism cannot be observed. Depending on the model, restricting access to shared mutable state might involve disallowing sharing entirely [25], only allowing single assignments to shared references [31, 2, 7], allowing sharing only by a limited form of message passing [15], ensuring that concurrent accesses to shared state are disjoint [5], resolving conflicting updates after the fact [17], or some combination of these approaches. These constraints can be imposed at the language or API level, within a type system, or at runtime.

5.1 Deterministic Parallel Java (DPJ)

DPJ [5, 4] is a deterministic language consisting of a system of annotations for Java code. A sophisticated region-based type system ensures that a mutable region of the heap is, essentially, passed linearly to an exclusive writer, thereby ensuring that the state accessed by concurrent threads is disjoint. DPJ does, however, provide a way to unsafely assert that operations commute with one another (using the commuteswith form) to enable concurrent mutation.

The LVars model differs from DPJ in that it allows overlapping shared state between threads as the default. Moreover, since LVar effects are already commutative, we avoid the need for commuteswith annotations. Finally, it is worth noting that while in DPJ, commutativity annotations have to appear in application-level code, in LVish only the data-structure author needs to write trusted code. The application programmer can run untrusted code that still enjoys a (quasi-)determinism guarantee, because only (quasi-)deterministic programs can be expressed as LVish Par computations.

More recently, Bocchino et al. [6] proposed a type and effect system that allows for the incorporation of non-deterministic sections of code in DPJ. The goal here is different from ours: while they aim to support intentionally nondeterministic computations such as those arising from optimization problems like branch-and-bound search, the quasi-determinism in LVish arises as a result of schedule nondeterminism.
5.2 FlowPools

Prokopec et al. [26] recently proposed a data structure with an API closely related to LVars extended with freezing and handlers: a FlowPool is a bag (that is, a multiset) that allows concurrent insertions but forbids removals, a seal operation that forbids further updates, and combinators like foreach that invoke callbacks as data arrives in the pool. To retain determinism, the seal operation requires explicitly passing the expected bag size as an argument, and the program will raise an exception if the bag goes over the expected size.

While this interface has a flavor similar to that of LVars, it lacks the ability to detect quiescence, which is crucial for expressing algorithms like graph traversal, and the seal operation is awkward to use when the structure of data is not known in advance. By contrast, the freeze operation on LVars does not require such advance knowledge, but moves the model into the realm of quasi-determinism. Another important difference is the fact that LVars are data structure-generic: both our formalism and our library support an unlimited collection of data structures, whereas FlowPools are specialized to bags.

5.3 Concurrent Revisions

The Concurrent Revisions (CR) [17] programming model uses isolation types to distinguish regions of the heap shared by multiple mutators. Rather than enforcing exclusive access in the style of DPI, CR clones a copy of the state for each mutator, using a deterministic “merge function” for resolving conflicts in local copies 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”. Determinism is enforced by the programming model allowing the programmer to specify which of two writing threads should prevail, regardless of the order in which those writes arrive, and 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 [9] 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 our model (in which any two LVar states have a lub).

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.

5.4 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 [11] had a notion of monotonicity based on set inclusion, analogous to the store ordering used in the proof of determinism for the (IVar-based) CnC system [7]. More recently, Conway et al. [10] generalized Bloom to a more flexible lattice-parameterized system, Bloom^L, in a manner analogous to our generalization from IVars to LVars. 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. Another difference between Bloom^L and the LVars model is that the former does not have a notion of quasi-determinism. Finally, 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.
6 Research plan

Together with my collaborators, I have already completed substantial work towards my thesis:

- Formally defined a parallel calculus, $\lambda_{LVar}$, that demonstrates the basic LVars model (with put and get operations); implemented it in PLT Redex; and proved determinism for it \[21, 20\].

- Formally defined the LVish calculus (called $\lambda_{LVish}$ in this proposal), which extends $\lambda_{LVar}$ with freezing and event handlers; implemented it in PLT Redex; and proved quasi-determinism for it \[23, 22\].

- Implemented and released the LVish Haskell library based on the formal LVars model \[23\].

To complete the thesis, I plan to do the following:

- Formally define the semantics of $\texttt{bump}$ and add it to $\lambda_{LVish}$. Prove determinism for the subset of $\lambda_{LVish}$ that includes $\texttt{bump}$, but does not include freezing and event handlers (this proof should be a straightforward refactoring of the existing determinism proof for $\lambda_{LVar}$\[12\]). Update the existing quasi-determinism proof for $\lambda_{LVish}$ to account for $\texttt{bump}$. (Time estimate: 1 month.)

- Extend the definition of CvRDTs to include threshold reads. Define and prove a query consistency property for CvRDTs extended thusly. (Time estimate: 2 months.)

- Implement PN-Counters and 2P-Sets in the LVish library and release a new version of LVish with these extensions. Implement at least one application that makes use of these CRDT-inspired LVars. (Time estimate: 3 months.)

- Expand Kuper et al. \[23\] to an extended version for journal submission, including the above material on $\texttt{bump}$, write a new paper on my work integrating CRDTs and LVars, and integrate the material from these papers into the dissertation itself. (Time estimate: 3 months.)

This plan puts me on track to defend in September 2014.

\[12\] Indeed, the updated version of the proof will be simpler because $\lambda_{LVar}$ has simultaneous parallel reductions, requiring a large amount of proof bookkeeping with regard to renaming of store locations, whereas $\lambda_{LVish}$ models parallelism with interleaving (as is the standard approach) and therefore is easier to reason about.
References


