LATTICE-BASED DATA STRUCTURES FOR DETERMINISTIC PARALLEL AND DISTRIBUTED PROGRAMMING

Lindsey Kuper

Submitted to the faculty of the University Graduate School
in partial fulfillment of the requirements
for the degree
Doctor of Philosophy
in the School of Informatics and Computing
Indiana University
August 2014

Accepted by the Graduate Faculty, Indiana University, in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Ryan R. Newton, Ph.D.

Amr Sabry, Ph.D.

Lawrence S. Moss, Ph.D.

Chung-chieh Shan, Ph.D.

08/29/2014

Acknowledgements

TODO: Write acks.

TODO: Write dedication.

Abstract

TODO: Write abstract.

Contents

Chapter 1. Introduction 1

1.1. The deterministic-by-construction parallel programming landscape 2

1.2. Lattice-based, monotonic data structures as a basis for deterministic parallelism 5

1.3. Quasi-deterministic and event-driven programming with LVars 6

1.4. The LVish library 8

1.5. Deterministic threshold queries of distributed data structures 10

1.6. Thesis statement, and organization of the rest of this dissertation 11

1.7. Previously published work 13

Chapter 2. LVars: lattice-based data structures for deterministic parallelism 14

2.1. Motivating example: a parallel, pipelined graph computation 17

2.2. Lattices, stores, and determinism 21

2.3. λ_{LVar}: Syntax and Semantics 28

2.4. Proof of Determinism for λ_{LVar} 34

2.5. Safe, Limited Nondeterminism 39

2.6. Conclusion 44

Chapter 3. Quasi-Deterministic and Event-Driven Programming with LVars 45

3.1. Introduction 45

3.2. Background: the LVars Model 51

3.3. LVish, Informally 56

3.4. LVish, Formally 60
3.5. Quasi-Determinism for LVish 70

Chapter 4. The LVish Library: Implementation and Evaluation 74
4.1. Prototype Implementation and Evaluation 74
4.2. Implementation 78
4.3. Evaluation: k-CFA Case Study 90

Chapter 5. Deterministic Threshold Queries of Distributed Data Structures 98
5.1. Introduction 98
5.2. How threshold queries work 104
5.3. Background: CvRDTs and eventual consistency 109
5.4. Adding threshold queries to CvRDTs 112
5.5. Determinism of threshold queries 119
5.6. Related work 120
5.7. Conclusion 121
5.8. Proof of Theorem 2 122

Chapter 6. Related Work 126
6.1. Related Work 131

Bibliography 136

CHAPTER 1

Introduction

Parallel programming—that is, writing programs that can take advantage of parallel hardware to go faster—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. While there are many ways to construct deterministic parallel programs and to verify the determinism of individual programs, only a deterministic-by-construction programming model can provide a *language-level* guarantee of determinism: deterministic programs are the only programs that can be expressed within the model.

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 want to count scheduling behavior as observable. In fact, we want to specifically allow tasks to be scheduled dynamically and unpredictably, without allowing such *schedule nondeterminism* to affect the observable behavior of a program. Therefore, we define the observable behavior of a program to be *the value to which the program evaluates.*

LK: In my proposal, I had a footnote here: "We assume that programs have no side effects other than state effects." I think I instead just want to say that we *ignore* other side effects. They can *happen*; it's just that they don't count.

Our definition of observable behavior ignores side effects other than *state*. Even under this limited definition of what is observable, though, sharing of state between parallel computations raises the possibility of *race conditions* that allow schedule nondeterminism to be observed in the outcome of a program. For instance, if one computation writes 3 to a shared location while another writes 4, then a subsequent third computation that reads and returns the location's contents will nondeterministically return 3 or 4, depending on the order in which the first two computations ran. Therefore, if a parallel programming model is to guarantee determinism by construction, it must necessarily limit sharing of mutable state between parallel tasks in some way.

1.1. The deterministic-by-construction parallel programming landscape

There is long-standing work on deterministic-by-construction parallel programming models that limit sharing of state between tasks. Surveying the landscape of possibilities, we find that they include:

- *No-shared-state parallelism.* One classic approach to guaranteeing determinism in a parallel programming model is to allow *no* shared mutable state between tasks, forcing tasks to produce values independently. An example of no-shared-state parallelism is pure functional programming with function-level task parallelism, or *futures*—for instance, in Haskell programs that use the `par` and `pseq` combinators [**31**]. The key characteristic of this style of programming is lack of side effects: because programs don't have side effects, expressions can evaluate simultaneously without affecting the eventual value of the program. Also belonging in this category are parallel programming models based on *pure data parallelism*, such as Data Parallel Haskell [**43, 11**] or the River Trail API for JavaScript [**25**], each of which extend existing languages with *parallel array* data types and (observably) pure operations on them. LK: Does it make sense to say that DPH is observably pure? It does mutate arrays.

- *Data-flow parallelism.* In *Kahn process networks* (KPNs) [**27**], as well as in the more restricted *synchronous data flow* systems [**28**], a network of independent "computing stations" communicate with each other through first-in first-out (FIFO) queues, or *channels*. Reading data out of such a FIFO queue is a *blocking* operation: once an attempt to read has started, a computing station cannot do anything else until the data to be read is available. Each station computes a sequential, 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 [**24**].
- *Single-assignment parallelism.* 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* [**3**] and are a well-established mechanism for enforcing determinism in parallel settings: they have appeared in Concurrent ML as `SyncVars` [**45**]; in the Intel Concurrent Collections (abbreviated "CnC") system [**9**]; and have even been implemented in hardware in Cray MTA machines [**5**]. Although most of these uses incorporate IVars into already-nondeterministic programming environments, the *monad-par* Haskell library [**32**] uses IVars in a deterministic-by-construction setting, allowing user-created threads to communicate through IVars without requiring the `IO` monad. Rather, operations that read and write

IVars must run inside a `Par` monad, thus encapsulating them inside otherwise pure programs, and hence a program in which the only effects are `Par` effects is guaranteed to be deterministic.

- *Imperative disjoint parallelism.* Finally, yet another approach to guaranteeing determinism is to ensure that the state accessed by concurrent threads is *disjoint*.LK: This is the first place I've used the word "concurrent". Should I explain concurrency vs. parallelism in a footnote or something? Is it even a useful distinction in this context? Sophisticated permissions systems and type systems make it possible for imperative programs to mutate state in parallel, while guaranteeing that the same state is not accessed simultaneously by multiple threads. We refer to this style of programming as *imperative disjoint parallelism*, with Deterministic Parallel Java (DPJ) [**7, 6**] as a prominent example.

The four parallel programming models listed above—no-shared-state parallelism, data-flow parallelism, single-assignment parallelism, and imperative disjoint parallelism—all seem to compriseLK: "have"? "embody"? rather different mechanisms for exposing parallelism and for ensuring determinism. If we view these different programming models as a toolkit of unrelated choices, though, it is not clear how to proceed when we want to implement an application with multiple parallelizable components that are best suited to different programming models. For example, suppose we have an application in which we wish to use data-flow pipeline parallelism via FIFO queues, but also disjoint parallel mutation of arrays. It is not obvious how to compose two programming models that each only allow communication through a single type of shared data structure—or, if we do manage to compose them, whether or not the determinism guarantee of the individual models is preserved by their composition. Hence we seek a general, broadly-applicable model for deterministic parallel programming that is not tied to a particular data structure.

1.2. Lattice-based, monotonic data structures as a basis for deterministic parallelism

In KPNs and other data-flow models, communication takes place over blocking FIFO queues with ever-increasing *channel histories*, while in IVar-based programming models such as CnC and monad-par, a shared data store of blocking single-assignment memory locations grows monotonically. Hence *monotonic data structures*—data structures to which information can only be added and never removed—emerge as a common theme of guaranteed-deterministic programming models.[1]

In this dissertation, I 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*.[2] 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 Chapter 2):

- The `put` operation can only change an LVar's state in a way that is *monotonically increasing* with respect to the 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

[1]We will later refine this definition of monotonic data structures to mean data structures to which information can only be added and never removed, *and* for which the order in which information is added is not observable.TODO: Add appropriate forward reference here.

[2]As I will explain in Chapter 2, what I am calling a "lattice" here really need only be a *bounded join-semilattice* augmented with a greatest element $\top$.

from the LVar, where every two elements in the threshold set must have the lattice's greatest element $\top$ as their least upper bound.[3] 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` operations on LVars are the only side effects will have the same observable result in spite of parallel execution and schedule nondeterminism. LK: Maybe we don't need the next sentence at all; maybe it's covered by the "organization" bullet points below. As we will see in Chapter 2, no-shared-state parallelism, data-flow parallelism and single-assignment parallelism are all subsumed by the LVars programming model, and as we will see in Chapter 4LK: Maybe refer to a specific section?, imperative disjoint parallelism is compatible with LVars as well.

1.3. Quasi-deterministic and event-driven 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

[3]See Chapter 2 for a generalization of this definition.TODO: Figure out what section to link to for generalized threshold sets.

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). In Chapter 3, I describe two extensions to the basic LVars model that make such computations possible:

- First, I describe how to add a primitive operation `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 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 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, as I discuss below (and, in more detail, in Chapter 4LK: Refer to a specific section?).

1.4. The LVish library

To demonstrate the practicality of the LVars programming model, in Chapter 4 I will describe *LVish*,[4] a Haskell library for deterministic and quasi-deterministic programming with LVars.

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.[5] 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

[4]Available at `http://hackage.haskell.org/package/lvish`.

[5]The `Par` monad exposed by LVish generalizes the original `Par` monad exposed by the *monad-par* library (`http://hackage.haskell.org/package/monad-par`, described by Marlow *et al.* [**32**]), which

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.

In addition to discussing the implementation of the LVish library and introducing the above features with examples, Chapter 4 illustrates LVish through three case studies, drawn from my collaborators' and my experience using the LVish library, all of which make use of handlers and freezing:

- First, I 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.

allows determinism-preserving communication between threads, but only through IVars, rather than LVars.

- Second, I 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 contrast the LVish implementation with the original sequential implementation and give performance results.
- Third, I describe using LVish to parallelize *PhyBin* [**40**], a bioinformatics application for comparing genealogical histories (phylogenetic trees) that relies heavily on a parallel tree-edit distance algorithm [**50**]. 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 argue that these non-idempotent writes, which we call `bump` operations, preserve determinism as long as programs do not use `put` and `bump` on the same LVar, a property that can be statically enforced by the aforementioned effect specification system in LVish.

1.5. Deterministic threshold queries of distributed data structures

The LVars model is closely related to the concept of *conflict-free replicated data types* (CRDTs) [**48**] for enforcing *eventual consistency* [**53**] of replicated objects in a distributed system. In particular, *state-based* or *convergent* replicated data types, abbreviated as *CvRDTs* [**48, 47**], leverage the mathematical properties of join-semilattices to guarantee that all replicas of an object (for instance, in a distributed database) eventually agree.

Although CvRDTs are provably eventually consistent, queries of CvRDTs (unlike threshold reads of LVars) nevertheless allow inconsistent intermediate states of replicas to be observed. That is, if two replicas of a CvRDT object are updated independently, reads of those replicas may disagree until a (least-upper-bound) *merge* operation takes place.

Taking inspiration from LVar-style threshold reads, in Chapter 5 I show how to extend CvRDTs to support provably deterministic, *strongly consistent* queries using a mechanism called *threshold queries* (or, seen from another angle, I show how to extend threshold reads from a shared-memory setting to a distributed one). The threshold query technique generalizes to any lattice, and hence any CvRDT, and allows deterministic observations to be made of replicated objects before the replicas' states have converged. This work has immediate practical relevance since, while many real distributed database systems allow a mix of eventually consistent and strongly consistent queries, CvRDTs only support the former, and threshold queries extend the CvRDT model to support both.

1.6. Thesis statement, and organization of the rest of this dissertation

With the above background, I can state my thesis:LK: This format ripped off from Josh Dunfield.

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

The rest of this dissertation supports my thesis as follows:

- *Lattice-based data structures*: In Chapter 2, I formally define LVars and use them to define λ_{LVar}, a call-by-value parallel calculus with shared state, including a runnable version implemented in PLT Redex [**19**] for interactive experimentation. In Chapter 3, I extend λ_{LVar} to add support for event handlers and the `freeze` operation, calling the resulting language λ_{LVish}.[6]
- *general*: In Chapter 2, I 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 λ_{LVar} is parameterized by the choice of lattice. For example, a lattice of channel histories with a prefix ordering allows LVars to represent FIFO channels that implement a Kahn process network, whereas instantiating λ_{LVar} with a lattice with one "empty" state and multiple "full" states (where $\forall i.\ \textit{empty} < \textit{full}_i$) results in a parallel single-assignment language. Hence λ_{LVar} is actually a *family* of calculi, varying by choice of lattice.
- *practical*: In Chapter 4, I 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*: In Chapter 2, I defend the claim that the basic LVars model guarantees determinism by giving a proof of determinism for λ_{LVar}, including the aforementioned `put` and `get` operations on LVars.
- *quasi-deterministic*: In Chapter 3, I defend this claim by defining quasi-determinism and giving a proof of quasi-determinism for the full λ_{LVish} calculus with the additions of the `freeze` operation and event handlers.

[6]λ_{LVish} is the *LVish calculus* described in Kuper *et al.* [**39**]. In this dissertation, I call it λ_{LVish} to avoid confusion with LVish, the Haskell library I describe in Chapter 4.

- *distributed programming*: In Chapter **??**, I defend the claim that the LVars programming model is applicable to distributed programming by showing how to extend distributed replicated data structures with LVar-style threshold reads.

1.7. Previously published work

This dissertation draws heavily on the earlier work and writing appearing in the following papers, written jointly with several collaborators:LK: This exact wording stolen from Aaron Turon. maybe tweak it?

- Lindsey Kuper and Ryan R. Newton. 2013. LVars: lattice-based data structures for deterministic parallelism. In *Proceedings of the 2nd ACM SIGPLAN workshop on Functional high-performance computing* (FHPC '13).
- Lindsey Kuper, Aaron Turon, Neelakantan R. Krishnaswami, and Ryan R. Newton. 2014. Freeze after writing: quasi-deterministic parallel programming with LVars. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (POPL '14).
- Lindsey Kuper, Aaron Todd, Sam Tobin-Hochstadt, and Ryan R. Newton. 2014. Taming the parallel effect zoo: extensible deterministic parallelism with LVish. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI '14).
- Lindsey Kuper and Ryan R. Newton. Deterministic threshold queries of distributed data structures. Draft, 2014.TODO: Update this if the paper is accepted.

LK: Also, in individual chapters, say, "The material in this chapter is based on research done jointly with..." and cite the papers.

CHAPTER 2

LVars: lattice-based data structures for deterministic parallelism

Programs written using a *deterministic-by-construction* model of parallel computation are guaranteed to always produce the same observable results, offering programmers freedom from subtle, hard-to-reproduce nondeterministic bugs. While a number of popular languages and language extensions (*e.g.*, Cilk [**22**]LK: Any others?) *encourage* deterministic parallel programming, few of them guarantee determinism at the language level—that is, for *all* programs that can be written using the model.

Of the options available for parallel programming with a language-level determinism guarantee, perhaps the most mature and broadly available choice is pure functional programming with function-level task parallelism, or *futures*. For example, Haskell programs using futures by means of the `par` and `pseq` combinators can provide real speedups on practical programs while guaranteeing determinism [**31**].[1] Yet pure programming with futures is not ideal for all problems. Consider a *producer/consumer* computation in which producers and consumers can be scheduled onto separate processors, each able to keep their working sets in cache. Such a scenario enables *pipeline parallelism* and is common, for instance, in stream processing. But a clear separation of producers and consumers is difficult with futures, because whenever a consumer

[1]When programming with `par` and `pseq`, a language-level determinism guarantee obtains if user programs are written in the *Safe Haskell* [**51**] subset of Haskell (which is implemented in GHC Haskell by means of the `SafeHaskell` language pragma), and if they do not use the `IO` monad.

forces a future, if the future is not yet available, the consumer immediately switches roles to begin computing the value (as explored by Marlow *et al.* [**32**]).

Since pure programming with futures is a poor fit for producer/consumer computations, one might then turn to *stateful* deterministic parallel models. Shared state between computations allows the possibility for race conditions that introduce nondeterminism, so any parallel programming model that hopes to guarantee determinism must do something to tame sharing—that is, to restrict access to mutable state shared among concurrent computations. Systems such as Deterministic Parallel Java [**7, 6**], for instance, accomplish this by ensuring that the state accessed by concurrent threads is *disjoint*. Alternatively, a programming model might allow *data* to be shared, but limit the *operations* that can be performed on it to only those operations that commute with one another and thus can tolerate nondeterministic thread interleavings. In such a setting, although the order in which side-effecting operations occur can differ on multiple runs, a program will always produce the same observable result.[2]

In Kahn process networks (KPNs) [**27**] and other *data-flow parallel* models—which are the basis for deterministic stream-processing languages such as StreamIt [**24**]—communication among processes takes place over blocking FIFO queues with ever-increasing *channel histories*. Meanwhile, in *single-assignment* [**52**] or *IVar-based* [**3**] programming models, such as the Intel Concurrent Collections system (CnC) [**9**] and the *monad-par* Haskell library [**32**], a shared data store of blocking single-assignment memory locations grows monotonically. Hence *monotonic data structures*—data structures to which information can only be added and never removed, and for which the order in which information is added is not observable—emerge

[2]There are many ways to define what is observable about a program. As noted in Chapter 1, I define the observable behavior of a program to be the value to which it evaluates.

as a common theme of both data-flow and single-assignment parallel programming models.

Because state modifications that only add information and never destroy it can be structured to commute with one another and thereby avoid race conditions, it stands to reason that diverse deterministic parallel programming models would leverage the principle of monotonicity. Yet systems like StreamIt, CnC, and monad-par emerge independently, without recognition of their common basis. Moreover, since each one of these programming models is based on a single type of shared data structure, they lack *generality*: IVars or FIFO streams alone cannot support all producer/consumer applications, as I discuss in Section 2.1 below.

Instead of limiting ourselves to a single data type of data structure, then, I propose taking the more general notion of monotonic data structures as the starting point for a new model for deterministic parallelism. The model I propose generalizes IVars to *LVars*, thus named because the states an LVar can take on are elements of an application-specific *lattice*.[3] In this chapter, I define LVars and use them to define λ_{LVar}, a deterministic parallel calculus with shared state, based on the call-by-value λ-calculus. The λ_{LVar} language is general enough to subsume existing deterministic parallel languages because it is parameterized by the choice of lattice. For example, a lattice of channel histories with a prefix ordering allows LVars to represent FIFO channels that implement a Kahn process network, whereas instantiating λ_{LVar} with a lattice with "empty" and "full" states (where *empty* $<$ *full*) results in a parallel

[3]As I'll explain in Section **??**, this "lattice" need only be a *bounded join-semilattice* augmented with a greatest element $\top$, in which every two elements have a least upper bound but not necessarily a greatest lower bound. For brevity, I use the term "lattice" in place of "bounded join-semilattice with a designated greatest element".

single-assignment language. Different instantiations of the lattice result in a family of deterministic parallel languages.

The main technical result of this chapter is a proof of determinism for λ_{LVar} (Section **??**). A critical aspect of the proof is a "frame" property, expressed by the Independence lemma (Section **??**), that would *not* hold in a typical language with shared mutable state, but holds in our setting because of the monotonic semantics of LVars.

Because lattices are composable, any number of diverse monotonic data structures can be used together safely. Moreover, as long as a data structure presents the LVar interface, it is fine to use an existing, optimized concurrent data structure implementation; we need not rewrite the world's data structures to leverage the λ_{LVar} determinism result. I discuss how to formulate a few common data structures (pairs, arrays, FIFOs) as lattices (Sections **??** and **??**) and how to implement operations on them within the λ_{LVar} model.

The material in this chapter is based on research done jointly with Ryan Newton [**36, 35**].

2.1. Motivating example: a parallel, pipelined graph computation

What applications motivate going beyond IVars and FIFO streams? Consider applications in which independent subcomputations contribute information to shared data structures that change monotonically with respect to some order. Hindley-Milner type inference is one example: in a parallel type-inference algorithm, each type variable monotonically acquires information through unification (which can be represented as a lattice). Likewise, in control-flow analysis, the *set* of locations to which a variable refers monotonically *shrinks*. In logic programming, a parallel implementation of conjunction might asynchronously add information to a logic variable from different threads.

To illustrate the issues that arise in computations of this nature, we consider a specific problem, drawn from the domain of *graph algorithms*, where issues of ordering create a tension between parallelism and determinism:

- In a directed graph, find the connected component containing a vertex v, and compute a (possibly expensive) function f over all vertices in that component, making the set of results available asynchronously to other computations.

For example, in a directed graph representing user profiles on a social network and the connections between them, where v represents a particular profile, we might wish to find all (or the first k degrees of) profiles connected to v, then map a function f over each profile in that set in parallel.

This is a challenge problem for deterministic-by-construction parallel programming: existing parallel solutions [**1**] often use a nondeterministic traversal of the connected component (even though the final connected component is deterministic), and IVars and streams provide no obvious aid. For example, IVars cannot accumulate sets of visited nodes, nor can they be used as "mark bits" on visited nodes, since they can only be written once and not tested for emptiness. Streams, on the other hand, impose an excessively strict ordering for computing the unordered *set* of vertex labels in a connected component. Yet before considering *new* mechanisms, we must also ask if a purely functional program can do the job.

2.1.1. A purely functional attempt. Figure 1 gives a Haskell implementation of a *level-synchronized* breadth-first graph traversal that finds the connected component reachable from a starting vertex. Nodes at distance one from the starting vertex are

```
nbrs :: Graph → NodeLabel → Set NodeLabel
-- ‘nbrs g n’ is the neighbor nodes of node ‘n’ in graph ‘g’.

-- Traverse each level of the graph in parallel, maintaining at each
-- recursive step a set of nodes that have been seen and a set of
-- nodes left to process.
bf_traverse :: Graph → Set NodeLabel → Set NodeLabel → Set NodeLabel
bf_traverse g seen nu =
  if nu == {}
  then seen
  else let seen'  = union seen nu
           allNbr = parFold union (parMap (nbrs g) nu)
           nu'    = difference allNbr seen'
       in bf_traverse g seen' nu'

-- Next we traverse the connected component, starting with the vertex ‘profile0’:
ccmp = bf_traverse profiles {} {profile0}
result = parMap analyze ccmp
```

FIGURE 1. A purely functional Haskell program that maps the `analyze` function over the connected component of the `profiles` graph that is reachable from the node `profile0`. Although component discovery proceeds in parallel, results of `analyze` are not asynchronously available to other computations, inhibiting pipelining.

discovered—and set-unioned into the connected component—before nodes of distance two are considered. Level-synchronization is a popular strategy for parallel breadth-first graph traversal (see, for instance, the Parallel Boost Graph Library [**1**]), although it necessarily sacrifices some parallelism for determinism: parallel tasks cannot continue discovering nodes in the component (racing to visit neighbor vertices) before synchronizing with all other tasks at a given distance from the start.

Unfortunately, the code given in Figure 1 does not quite implement the problem specification given above. Even though connected-component discovery is parallel, members of the output set do not become available to other computations until component discovery is *finished*, limiting parallelism. We could manually push the `analyze` invocation inside the `bf_traverse` function, allowing the `analyze` computation to start sooner, but then we push the same problem to the downstream consumer, unless we are able to perform a heroic whole-program fusion. TODO: Emphasize the following point: we're prevented from PIPELINING the work here because we have no monotonicity guarantee. Kahn 1974 makes a point of talking about how in KPNs, monotonicity enables both pipelining and determinism.

If `bf_traverse` returned a list, lazy evaluation could make it possible to *stream* results to consumers incrementally. But since it instead returns a *set*, such pipelining is not generally possible: consuming the results early would create a proof obligation that the determinism of the consumer does not depend on the order in which results emerge from the producer.[4]

A compromise would be for `bf_traverse` to return a list of "level-sets": distance one nodes, distance two nodes, and so on. Thus level-one results could be consumed before level-two results are ready. Still, the problem would remain: within each level-set, we cannot launch all instances of `analyze` and asynchronously use those results that

[4]As intuition for this idea, consider that purely functional set data structures, such as Haskell's `Data.Set`, are typically represented with balanced trees. Unlike with lists, the structure of the tree is not known until all elements are present.

finish *first*. Moreover, we would still have to contend with the previously-mentioned difficulty of separating producers and consumers when expressing producer-consumer computations using pure programming with futures [**32**].

2.1.2. An LVar-based solution. Suppose that we could write a breadth-first traversal in a programming model with limited effects that allows *any* data structure to be shared among tasks, including sets and graphs, so long as that data structure grows *monotonically*. Consumers of the data structure may execute as soon as data is available, but may only observe irrevocable, monotonic properties of it. This is possible with a programming model based on LVars. In the rest of this chapter, I will formally introduce LVars and the λ_{LVar} language and give a proof of determinism for it. Later, in Section **??**, we will return to `bf_traverse` and see how to implement a version of it using LVars. TODO: Put a forward reference to chapter 4 here? Or maybe just do the solution in this chapter?

LK: Mostly edited up to this point.

2.2. Lattices, stores, and determinism

TODO: lambdaLVar should just be a subset of lambdaLVish. We don't need reify, reflect, consume, rename, or any of that.

As a minimal substrate for LVars, we introduce λ_{LVar}, a parallel call-by-value λ-calculus extended with a *store* and with communication primitives `put` and `get` that operate on data in the store. The class of programs that we are interested in modeling with λ_{LVar} are those with explicit effectful operations on shared data structures,

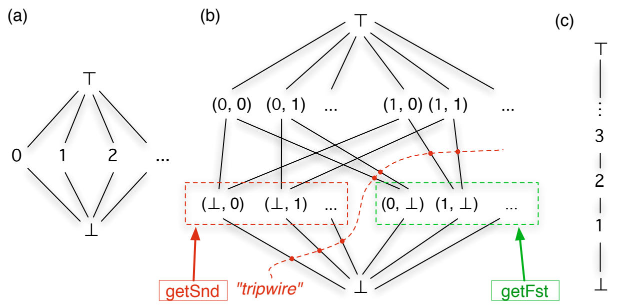

FIGURE 2. Example lattices: (a) IVar containing a natural number; (b) pair of natural-number-valued IVars; (c) positive integers ordered by $\leq$ (see Section 2.5.1). Subfigure (b) is annotated with example threshold sets that would correspond to a blocking read of the first or second element of the pair (see Sections 2.2.3 and 2.3.2). Any state transition crossing the "tripwire" for `getSnd` causes it to unblock and return a result.

in which subcomputations may communicate with each other via the `put` and `get` operations.

In this setting of shared mutable state, the trick that λ_{LVar} employs to maintain determinism is that stores contain *LVars*, which are a generalization of IVars.[5] Whereas IVars are single-assignment variables—either empty or filled with an immutable value—an LVar may have an arbitrary number of states forming a set D, which is partially ordered by a relation $\sqsubseteq$. An LVar can take on any sequence of states from D, so long as that sequence respects the partial order—that is, updates to the LVar (made via the `put` operation) are *inflationary* with respect to $\sqsubseteq$. Moreover, the `get` operation allows only limited observations of the LVar's state. In this section, we discuss how lattices and stores work in λ_{LVar} and explain how the semantics of `put` and `get` together enforce determinism in λ_{LVar} programs.

[5]IVars are so named because they are a special case of *I-structures* [**3**]—namely, those with only one cell.

2.2.1. Lattices. The definition of λ_{LVar} is parameterized by the choice of D: to write concrete λ_{LVar} programs, one must specify the set of LVar states that one is interested in working with, and an ordering on those states. Therefore λ_{LVar} is actually a *family* of languages, rather than a single language.

Formally, D is a *bounded join-semilattice* augmented with a greatest element $\top$. That is, D has the following structure:

- D has a least element $\bot$, representing the initial "empty" state of an LVar.
- D has a greatest element $\top$, representing the "error" state that results from conflicting updates to an LVar.
- D comes equipped with a partial order $\sqsubseteq$, where $\bot \sqsubseteq d \sqsubseteq \top$ for all $d \in D$.
- Every pair of elements in D has a least upper bound (lub) $\sqcup$. Intuitively, the existence of a lub for every two elements in 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.

Virtually any data structure to which information is added gradually can be represented as a lattice, including pairs, arrays, trees, maps, and infinite streams. In the case of maps or sets, $\sqcup$ could be defined simply as union; for pointer-based data structures like tries, it could allow for unification of partially-initialized structures.

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(a), that is,

$$D = (\{\top, \bot\} \cup \mathbb{N}, \sqsubseteq),$$

where the partial order $\sqsubseteq$ 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(c), on the other hand, the $\top$ state 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, the resulting state will be 2, since $2 \sqcup 1 = 2$. Here, the unreachability of $\top$ models the fact that no conflicting updates can occur to the LVar.

2.2.2. Stores. During the evaluation of a λ_{LVar} program, 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 *Loc*, to its state, which is some element $d \in D$. Although each LVar in a program has its own state, the states of all the LVars are drawn from the same lattice D. We can do this with no loss of generality because lattices corresponding to different types of LVars could always be unioned into a single lattice (with shared $\top$ and $\bot$ elements). Alternatively, in a typed formulation of λ_{LVar}, the type of an LVar might determine the lattice of its states.

Definition 1. A *store* is either a finite partial mapping $S : Loc \overset{\text{fin}}{\to} (D - \{\top\})$, or the distinguished element $\top_S$.

We use the notation $S[l \mapsto (d,)]$ to denote extending S with a binding from l to d. If $l \in dom(S)$, then $S[l \mapsto (d,)]$ 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 $[l_1 \mapsto (d_1, ,) \ldots, l_n \mapsto (d_n,)]$. The state space of stores forms a bounded join-semilattice augmented with a greatest element, just as D does, with the empty store $\bot_S$ as its least element and $\top_S$ as its greatest element. It is straightforward to lift the $\sqsubseteq$ and $\sqcup$ operations defined on elements of D to the level of stores:

Definition 2. A store S is *less than or equal to* a store S' (written $S \sqsubseteq_S S'$) iff:

- $S' = \top_S$, or
- $dom(S) \subseteq dom(S')$ and for all $l \in dom(S)$, $S(l) \sqsubseteq_p S'(l)$.

Definition 3. The *least upper bound (lub)* of two stores S_1 and S_2 (written $S_1 \sqcup_S S_2$) is defined as follows:

- $S_1 \sqcup_S S_2 = \top_S$ iff $S_1 = \top_S$ or $S_2 = \top_S$.
- $S_1 \sqcup_S S_2 = \top_S$ iff there exists some $l \in dom(S_1) \cap dom(S_2)$ such that $S_1(l) \sqcup_p S_2(l) = \top_p$.
- Otherwise, $S_1 \sqcup_S S_2$ is the store S such that:
  - $dom(S) = dom(S_1) \cup dom(S_2)$, and
  - For all $l \in dom(S)$:

$$S(l) = \begin{cases} S_1(l) \sqcup_p S_2(l) & \text{if } l \in dom(S_1) \cap dom(S_2) \\ S_1(l) & \text{if } l \notin dom(S_2) \\ S_2(l) & \text{if } l \notin dom(S_1) \end{cases}$$

By Definition **??**, if $d_1 \sqcup d_2 = \top$, then $[l \mapsto (d_1,)] \sqcup_S [l \mapsto (d_2,)] = \top_S$. Notice that a store containing a binding $l \mapsto (\top,)$ can never arise during the execution of a λ_{LVar} program, because (as we will see in Section 2.3) an attempted write that would take the state of l to $\top$ would raise an error before the write can occur.

2.2.3. Communication Primitives. The `new`, `put`, and `get` operations create, write to, and read from LVars, respectively. The interface is similar to that presented by mutable references:

- `new` extends the store with a binding for a new LVar whose initial state is $\bot$, and returns the location l of that LVar (*i.e.*, a pointer to the LVar).
- `put` takes a pointer to an LVar and a singleton set containing a new state and updates the LVar's state to the *least upper bound* of the current state and the new state, potentially pushing the state of the LVar upward in the lattice. Any update that would take the state of an LVar to $\top$ results in an error.
- `get` performs a blocking "threshold" read that allows limited observations of the state of an LVar. It takes a pointer to an LVar and a *threshold set* Q, which is a non-empty subset of D that is *pairwise incompatible*, meaning that the lub of any two distinct elements in Q is $\top$. If the LVar's state d in the lattice is *at or above* some $d' \in Q$, the `get` operation unblocks and returns the singleton set $\{d'\}$. Note that d' is a unique element of Q, for if there is another $d'' \neq d'$ in the threshold set such that $d'' \sqsubseteq d$, it would follow that $d' \sqcup d'' \sqsubseteq d \neq \top$, which contradicts the requirement that Q be pairwise incompatible.

The intuition behind `get` is that it specifies a subset of the lattice that is "horizontal": no two elements in the threshold set can be above or below one another. Intuitively, each element in the threshold set is an "alarm" that detects the activation of itself or any state above it. One way of visualizing the threshold set for a `get` operation is as a subset of edges in the lattice that, if crossed, set off the corresponding alarm. Together these edges form a "tripwire". This visualization is pictured in Figure 1(b). The threshold set $\{(\bot, 0), (\bot, 1), ...\}$ (or a subset thereof) would pass the incompatibility test, as would the threshold set $\{(0, \bot), (1, \bot), ...\}$ (or a subset thereof), but a combination of the two would not pass.

Both `get` and `put` take and return *sets*. The fact that `put` takes a singleton set and `get` returns a singleton set (rather than a value d) may seem awkward; it is merely a way to keep the grammar for values simple, and avoid including set primitives in the language (*e.g.*, for converting d to $\{d\}$).

2.2.4. Monotonic Store Growth and Determinism. In IVar-based languages, a store can only change in one of two ways: a new binding is added at $\bot$, or a previously $\bot$ binding is permanently updated to a meaningful value. It is therefore straightforward in such languages to define an ordering on stores and establish determinism based on the fact that stores grow monotonically with respect to the ordering. For instance, *Featherweight CnC* [**9**], a single-assignment imperative calculus that models the Intel Concurrent Collections (CnC) system, defines ordering on stores as follows:[6]

DEFINITION 4 (store ordering, Featherweight CnC). A store S is *less than or equal to* a store S' (written $S \sqsubseteq_S S'$) iff $dom(S) \subseteq dom(S')$ and for all $l \in dom(S)$, $S(l) = S'(l)$.

Our Definition 8 is reminiscent of Definition 4, but Definition 4 requires that $S(l)$ and $S'(l)$ be *equal*, instead of our weaker requirement that $S(l) \sqsubseteq S'(l)$ according to the user-provided partial order $\sqsubseteq$. In λ_{LVar}, stores may grow by updating existing bindings via repeated `puts`, so Definition 4 would be too strong; for instance, if $\bot \sqsubset d_1 \sqsubseteq d_2$ for distinct $d_1, d_2 \in D$, the relationship $[l \mapsto (d_1,)] \sqsubseteq_S [l \mapsto (d_2,)]$ holds under Definition 8, but would not hold under Definition 4. That is, in λ_{LVar} an LVar could take on the state d_1 followed by d_2, which would not be possible in Featherweight CnC. We establish in Section 3.5 that λ_{LVar} remains deterministic despite the relatively weak $\sqsubseteq_S$ relation given in Definition 8. The keys to maintaining determinism are the blocking semantics of the `get` operation and the fact that it allows only *limited* observations of the state of an LVar.

[6]In Featherweight CnC, no store location is explicitly bound to $\bot$. Instead, if $l \notin dom(S)$ then l is defined to be at $\bot$.

2.3. $\lambda_{\mathbf{LVar}}$: Syntax and Semantics

The syntax and operational semantics of λ_{LVar} appear in Figures **??** and **??**, respectively.[7] As we have noted, both the syntax and semantics are parameterized by the lattice D. The reduction relation $\hookrightarrow$ is defined on *configurations* $\langle S;\, e\rangle$ comprising a store and an expression. The *error configuration*, written **error**, is a unique element added to the set of configurations, but we consider $\langle \top_S;\, e\rangle$ to be equal to **error**, for all expressions e. The metavariable σ ranges over configurations.

Figure **??** shows two disjoint sets of reduction rules: those that step to configurations other than **error**, and those that step to **error**. Most of the latter set of rules merely propagate errors along. A new **error** can only arise by way of the E-ParAppErr rule, which represents the joining of two conflicting subcomputations, or by way of the E-PutValErr rule, which applies when a `put` to a location would take its state to $\top$.

The rules E-New, E-PutVal/E-PutValErr, and E-GetVal respectively express the semantics of the `new`, `put`, and `get` operations described in Section 2.2.3. The incompatibility property of the threshold set argument to `get` is enforced in the E-GetVal rule by the $\mathit{incomp}(Q)$ premise, which requires that the least upper bound of any two distinct elements in Q must be $\top$.

The E-Put-1/E-Put-2 and E-Get-1/E-Get-2 rules allow for reduction of subexpressions inside `put` and `get` expressions until their arguments have been evaluated, at which time the E-PutVal (or E-PutValErr) and E-GetVal rules respectively apply. Arguments to `put` and `get` are evaluated in arbitrary order, although not

[7] We have implemented λ_{LVar} as a runnable PLT Redex [**19**] model, available in the LVars repository.

Given a lattice $(D, \sqsubseteq, \bot, \top)$ with elements $d \in D$:

$$\begin{array}{rrl}
\text{configurations} & \sigma & ::= \langle S;\, e\rangle \;\mid\; \textsf{error} \\
\text{expressions} & e & ::= x \mid v \mid e\ e \mid \texttt{get}\ e\ e \mid \texttt{put}\ e\ e \mid \texttt{new} \mid \texttt{freeze}\ e \\
 & & \quad\mid \texttt{freeze}\ e\ \texttt{after}\ e\ \texttt{with}\ e \\
 & & \quad\mid \texttt{freeze}\ l\ \texttt{after}\ Q\ \texttt{with}\ \lambda x.\, e, \{e, \dots\}, H \\
\text{stores} & S & ::= [l_1 \mapsto p_1, \dots, l_n \mapsto p_n] \mid \top_S \\
\text{values} & v & ::= \texttt{()} \mid d \mid p \mid l \mid P \mid Q \mid \lambda x.\, e \\
\text{eval contexts} & E & ::= [\,] \mid E\ e \mid e\ E \mid \texttt{get}\ E\ e \mid \texttt{get}\ e\ E \mid \texttt{put}\ E\ e \\
 & & \quad\mid \texttt{put}\ e\ E \mid \texttt{freeze}\ E \mid \texttt{freeze}\ E\ \texttt{after}\ e\ \texttt{with}\ e \\
 & & \quad\mid \texttt{freeze}\ e\ \texttt{after}\ E\ \texttt{with}\ e \mid \texttt{freeze}\ e\ \texttt{after}\ e\ \texttt{with}\ E \\
 & & \quad\mid \texttt{freeze}\ v\ \texttt{after}\ v\ \texttt{with}\ v, \{e \dots\ E\ e \dots\}, H
\end{array}$$

$$\begin{array}{rrl}
\text{``handled'' sets} & H & ::= \{d_1, \dots, d_n\} \\
\text{threshold sets} & P & ::= \{p_1, p_2, \dots\} \\
\text{event sets} & Q & ::= \{d_1, d_2, \dots\}
\end{array}
\qquad
\begin{array}{rrl}
\text{states} & p & ::= (d, \mathit{frz}) \\
\text{status bits} & \mathit{frz} & ::= \textsf{true} \mid \textsf{false}
\end{array}$$

FIGURE 3. Syntax for λ_{LVish}.

simultaneously, for simplicity's sake. However, it would be straightforward to add E-ParPut and E-ParGet rules to the semantics that are analogous to E-ParApp, should simultaneous evaluation of `put` and `get` arguments be desired.

LK: TODO: we need a new figure that's λ_{LVish} but with the freezing, etc., removed.

2.3.1. Fork-Join Parallelism. λ_{LVar} has an explicitly parallel reduction semantics: the E-ParApp rule in Figure ?? allows simultaneous reduction of the operator and operand in an application expression, so that (eliding stores) the application $e_1\ e_2$ may step to $e_1'\ e_2'$ if e_1 steps to e_1' and e_2 steps to e_2'. In the case where one of the subexpressions is already a value or is otherwise unable to step (for instance, if it is a blocked `get`), the reflexive E-Refl rule comes in handy: it allows the E-ParApp rule to apply nevertheless. When the configuration $\langle S;\, e_1\ e_2 \rangle$ takes a step, e_1 and e_2 step as separate subcomputations, each beginning with its own copy of the store S. Each subcomputation can update S independently, and we combine the resulting two stores by taking their least upper bound when the subcomputations rejoin. (Because E-ParApp and E-ParAppErr perform truly simultaneous reduction, they have to address the subtle point of location renaming: locations created while e_1 steps must be renamed to avoid name conflicts with locations created while e_2 steps. We discuss the *rename* metafunction and other issues related to renaming in Appendix **??**.)

Although the semantics admits such parallel reductions, λ_{LVar} is still call-by-value in the sense that arguments must be fully evaluated before function application (β-reduction, modeled by the E-Beta rule) can occur. We can exploit this property to define a syntactic sugar `let par` for *parallel composition*, which computes two subexpressions e_1 and e_2 in parallel before computing e_3:

$$\begin{array}{rcl} \texttt{let par}\ x = e_1 & & \\ y = e_2 & \stackrel{\triangle}{=} & ((\lambda x.\, (\lambda y.\, e_3))\ e_1)\ e_2 \\ \texttt{in}\ e_3 & & \end{array}$$

Although e_1 and e_2 can be evaluated in parallel, e_3 cannot be evaluated until both e_1 and e_2 are values, because the call-by-value semantics does not allow β-reduction until the operand is fully evaluated, and because it further disallows reduction under λ-terms (sometimes called "full β-reduction"). In the terminology of parallel programming, a `let par` expression executes both a *fork* and a *join*. Indeed, it is common for fork and join to be combined in a single language construct, for example, in languages with parallel tuple expressions such as Manticore [**20**].

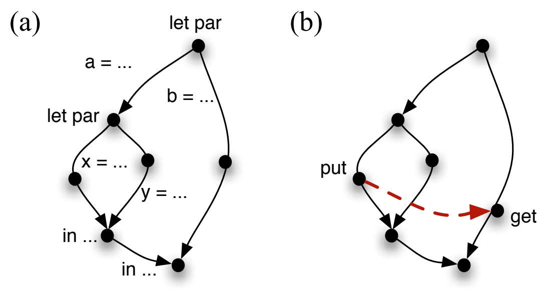

FIGURE 4. A series-parallel graph induced by parallel λ-calculus evaluation (a); a non-series-parallel graph induced by `put`/`get` operations (b).

Since `let par` expresses *fork-join* parallelism, the evaluation of a program comprising nested `let par` expressions would induce a runtime dependence graph like that pictured in Figure 4(a). The λ_{LVar} language (minus `put` and `get`) can support any *series-parallel* dependence graph. Adding communication through `put` and `get` introduces "lateral" edges between branches of a parallel computation, as in Figure 4(b). This adds the ability to construct arbitrary non-series-parallel dependency graphs, just as with *first-class futures* [**49**].

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

2.3.2. Programming with `put` and `get`. RRN: Perhaps it's not important, but we don't seem to specify whether $\sqsubseteq$ on the numeric components of the pair is an IVar style ordering (horizontal), or the normal $\leq$ on numbers. It's potentially confusing that we use $\leq$ in the intro, but then never use that order again.

For our first example of a λ_{LVar} program, we choose the elements of our lattice to be pairs of natural-number-valued IVars, as shown in Figure 1(b). We can then write the following program:

(Example 1)
$$\begin{array}{l}\texttt{let}\ p = \texttt{new in}\\ \quad \texttt{let}\ \_ = \texttt{put}\ p\ \{(3,4)\}\ \texttt{in}\\ \quad\quad \texttt{let}\ v_1 = \texttt{get}\ p\ \{(\bot, n) \mid n \in \mathbb{N}\}\ \texttt{in}\\ \quad\quad\quad \ldots v_1 \ldots\end{array}$$

This program creates a new LVar p and stores the pair $(3,4)$ in it. $(3,4)$ then becomes the *state* of p. The premises of the E-GetVal reduction rule hold: $S(p) = (3,4)$; the threshold set $Q = \{(\bot, n) \mid n \in \mathbb{N}\}$ is a pairwise incompatible subset of D; and there exists an element $d_1 \in Q$ such that $d_1 \sqsubseteq (3,4)$. In particular, the pair $(\bot, 4)$ is a member of Q, and $(\bot, 4) \sqsubseteq (3,4)$. Therefore, `get` p $\{(\bot, n) \mid n \in \mathbb{N}\}$ returns the singleton set $\{(\bot, 4)\}$, which is a first-class value in λ_{LVar} that can, for example, subsequently be passed to `put`.

Since threshold sets can be cumbersome to read, we can define some convenient shorthands `getFst` and `getSnd` for working with our lattice of pairs:

$$\texttt{getFst}\ p \triangleq \texttt{get}\ p\ \{(n, \bot) \mid n \in \mathbb{N}\}$$
$$\texttt{getSnd}\ p \triangleq \texttt{get}\ p\ \{(\bot, n) \mid n \in \mathbb{N}\}$$

The approach we take here for pairs generalizes to arrays of arbitrary size, with *streams* being the special case of unbounded arrays where consecutive locations are written.

Querying incomplete data structures. It is worth noting that `getSnd` p returns a value even if the first entry of p is not filled in. For example, if the `put` in the second line of (Example 1) had been `put` p $\{(\bot, 4)\}$, the `get` expression would still return $\{(\bot, 4)\}$. It is therefore possible to safely query an incomplete data structure—say, an object that is in the process of being initialized by a constructor. However, notice that we *cannot* define a `getFstOrSnd` function that returns if either entry of a pair is filled in. Doing so would amount to passing all of the boxed elements of the lattice in Figure 1(b) to `get` as a single threshold set, which would fail to satisfiy the incompatibility criterion.

RRN: This has become somewhat redundant since it has been pointed out elsewhere... maybe it could be axed to save a few lines.

Blocking reads. On the other hand, consider the following:

(Example 2)
$$
\begin{array}{l}
\texttt{let } p = \texttt{new in} \\
\quad \texttt{let } \_ = \texttt{put } p \ \{(\bot, 4)\} \texttt{ in} \\
\quad\quad \texttt{let par } v_1 = \texttt{getFst } p \\
\quad\quad\quad\quad\quad\quad \_ \ = \texttt{put } p \ \{(3, 4)\} \\
\quad\quad \texttt{in } \ldots \ v_1 \ \ldots
\end{array}
$$

Here `getFst` can attempt to read from the first entry of p before it has been written to. However, thanks to `let par`, the `getFst` operation is being evaluated in parallel with a `put` operation that will give it a value to read, so `getFst` simply *blocks* until `put` p $\{(3, 4)\}$ has been evaluated, at which point the evaluation of `getFst` p can proceed.

In the operational semantics, this blocking behavior corresponds to the last premise of the E-GetVal rule not being satisfied. In (Example 2), although the threshold set $\{(n, \bot) \mid n \in \mathbb{N}\}$ is incompatible, the E-GetVal rule cannot apply because there is no state in the threshold set that is lower than the state of p in the lattice—that is, we are trying to `get` something that is not yet there! It is only after p's state is updated that the premise is satisfied and the rule applies.

By induction hypothesis, there exist σ_{c_1}, σ_{c_2} such that

$$\begin{tikzcd}
& \langle S\,;\, e_1\rangle \arrow[dl] \arrow[dr] & \\
\langle S_{a_1};\, e_{a_1}\rangle \arrow[dr] & & \langle S_{b_1};\, e_{b_1}\rangle \arrow[dl] \\
& \sigma_{c_1}\ (= \langle S_{c_1};\, e_{c_1}\rangle \text{ or } \textbf{error}) &
\end{tikzcd}
\qquad
\begin{tikzcd}
& \langle S\,;\, e_2\rangle \arrow[dl] \arrow[dr] & \\
\langle S_{a_2};\, e_{a_2}\rangle \arrow[dr] & & \langle S_{b_2};\, e_{b_2}\rangle \arrow[dl] \\
& \sigma_{c_2}\ (= \langle S_{c_2};\, e_{c_2}\rangle \text{ or } \textbf{error}) &
\end{tikzcd}$$

To show: There exists σ_c such that

$$\begin{tikzcd}
& \langle S\,;\, e_1\ e_2\rangle \arrow[dl] \arrow[dr] & \\
\langle S_{a_1} \sqcup_S S_{a_2}\,;\, e_{a_1}\ e_{a_2}\rangle \arrow[dr] & & \langle S_{b_1} \sqcup_S S_{b_2}\,;\, e_{b_1}\ e_{b_2}\rangle \arrow[dl] \\
& \sigma_c &
\end{tikzcd}$$

FIGURE 5. Diagram of the subcase of Lemma 4 in which the E-PARAPP rule is the last rule in the derivation of both $\sigma \hookrightarrow \sigma_a$ and $\sigma \hookrightarrow \sigma_b$. We are required to show that, if the configuration $\langle S;\, e_1\ e_2\rangle$ steps by E-PARAPP to two different configurations, $\langle S_{a_1} \sqcup_S S_{a_2};\, e_{a_1}\ e_{a_2}\rangle$ and $\langle S_{b_1} \sqcup_S S_{b_2};\, e_{b_1}\ e_{b_2}\rangle$, then they both step to some third configuration σ_c.

2.4. Proof of Determinism for $\lambda_{\mathbf{LVar}}$

Our main technical result is a proof of determinism for the λ_{LVar} language. Most proofs are only sketched here; the complete proofs appear in the companion technical report [**?**].

2.4.1. Supporting Lemmas. Figure 6 shows a *frame rule*, due to O'Hearn *et al.* [**42**], which captures the idea that, given a program c with a precondition p that holds before it runs and a postcondition q that holds afterward, a disjoint condition r that holds before c runs will continue to hold afterward. Moreover, the original postcondition q will continue to hold. For λ_{LVar}, we can state and prove an analogous *local reasoning* property. Lemma 1, the Independence lemma, says that if the configuration $\langle S;\, e\rangle$ can step to $\langle S';\, e'\rangle$, then the configuration $\langle S \sqcup_S S'';\, e\rangle$, where S'' is some other store (*e.g.*, one from another subcomputation), can step to $\langle S' \sqcup_S S'';\, e'\rangle$. Roughly speaking, the Independence lemma allows us to "frame on" a larger store around S and still finish the transition with the original result e', which is key to being able to carry out the determinism proof.

Frame rule (O'Hearn *et al.*, 2001):

$$\frac{\{p\}\ c\ \{q\}}{\{p * r\}\ c\ \{q * r\}}\ \text{(where no free variable in } r \text{ is modified by } c\text{)}$$

Lemma 1 (Independence), simplified:

$$\frac{\langle S;\, e\rangle \hookrightarrow \langle S';\, e'\rangle}{\langle S \sqcup_S S'';\, e\rangle \hookrightarrow \langle S' \sqcup_S S'';\, e'\rangle}\ \begin{array}{l}(S'' \text{ non-conflicting with} \\ \langle S;\, e\rangle \hookrightarrow \langle S';\, e'\rangle)\end{array}$$

FIGURE 6. Comparison of a standard frame rule with a simplified version of the Independence lemma. The $*$ connective in the frame rule requires that its arguments be disjoint. The Independence lemma generalizes $*$ to least upper bound.

LEMMA 1 (Independence). *If $\langle S;\, e\rangle \hookrightarrow \langle S';\, e'\rangle$ (where $\langle S';\, e'\rangle \neq$ **error**), then for all S'' such that S'' is non-conflicting with $\langle S;\, e\rangle \hookrightarrow \langle S';\, e'\rangle$ and $S' \sqcup_S S'' \neq \top_S$:*

$\langle S \sqcup_S S'';\, e\rangle \hookrightarrow \langle S' \sqcup_S S'';\, e'\rangle$.

PROOF SKETCH. By induction on the derivation of $\langle S;\, e\rangle \hookrightarrow \langle S';\, e'\rangle$, by cases on the last rule in the derivation. □

The Clash lemma, Lemma 2, is similar to the Independence lemma, but handles the case where $S' \sqcup_S S'' = \top_S$. It ensures that in that case, $\langle S \sqcup_S S'';\, e\rangle$ steps to **error**.

LEMMA 2 (Clash). *If $\langle S;\, e\rangle \hookrightarrow \langle S';\, e'\rangle$ (where $\langle S';\, e'\rangle \neq$ **error**), then for all S'' such that S'' is non-conflicting with $\langle S;\, e\rangle \hookrightarrow \langle S';\, e'\rangle$ and $S' \sqcup_S S'' = \top_S$:*

*$\langle S \sqcup_S S'';\, e\rangle \hookrightarrow$ **error**.*

PROOF SKETCH. By induction on the derivation of $\langle S;\, e\rangle \hookrightarrow \langle S';\, e'\rangle$, by cases on the last rule in the derivation. □

Finally, Lemma 3 says that if a configuration $\langle S;\, e\rangle$ steps to **error**, then evaluating e in some larger store will also result in **error**.

LEMMA 3 (Error Preservation). *If* $\langle S;\, e\rangle \hookrightarrow \textbf{error}$ *and* $S \sqsubseteq_S S'$*, then* $\langle S';\, e\rangle \hookrightarrow \textbf{error}$.

PROOF SKETCH. By induction on the derivation of $\langle S;\, e\rangle \hookrightarrow \textbf{error}$, by cases on the last rule in the derivation. □

Non-conflicting stores. In the Independence and Clash lemmas, S'' must be *non-conflicting* with the original transition $\langle S;\, e\rangle \hookrightarrow \langle S';\, e'\rangle$. We say that a store S'' is *non-conflicting* with a transition $\langle S;\, e\rangle \hookrightarrow \langle S';\, e'\rangle$ iff $dom(S'')$ does not have any elements in common with $dom(S') - dom(S)$, which is the set of names of *new* store bindings created between $\langle S;\, e\rangle$ and $\langle S';\, e'\rangle$.

DEFINITION 5. A store S'' is *non-conflicting* with the transition $\langle S;\, e\rangle \hookrightarrow \langle S';\, e'\rangle$ iff $(dom(S') - dom(S)) \cap dom(S'') = \emptyset$.

The purpose of the non-conflicting requirement is to rule out location name conflicts caused by allocation. It is possible to meet this non-conflicting requirement by applying the *rename* metafunction, which we define and prove the safety of in Appendix **??**.

Requiring that a store S'' be non-conflicting with a transition $\langle S;\, e\rangle \hookrightarrow \langle S';\, e'\rangle$ is not as restrictive a requirement as it appears to be at first glance: it is fine for S'' to contain bindings for locations that are bound in S', as long as they are also locations bound in S. In fact, they may even be locations that were *updated* in the transition from $\langle S;\, e\rangle$ to $\langle S';\, e'\rangle$, as long as they were not *created* during that transition. In other words, given a store S'' that is non-conflicting with $\langle S;\, e\rangle \hookrightarrow \langle S';\, e'\rangle$, it may still be the case that $dom(S'')$ has elements in common with $dom(S)$, and with the subset of $dom(S')$ that is $dom(S)$.

2.4.2. Diamond Lemma. Lemma 4 does the heavy lifting of our determinism proof: it establishes the *diamond property*, which says that if a configuration steps to two different configurations, there exists a single third configuration to which those configurations both step. Here, again, we rely on the ability to safely rename locations in a configuration, as discussed in Appendix **??**.

LEMMA 4 (Diamond). *If $\sigma \hookrightarrow \sigma_a$ and $\sigma \hookrightarrow \sigma_b$, then there exists σ_c such that either:*

- *$\sigma_a \hookrightarrow \sigma_c$ and $\sigma_b \hookrightarrow \sigma_c$, or*
- *there exists a safe renaming σ_b' of σ_b with respect to $\sigma \hookrightarrow \sigma_b$, such that $\sigma_a \hookrightarrow \sigma_c$ and $\sigma_b' \hookrightarrow \sigma_c$.*

PROOF SKETCH. By induction on the derivation of $\sigma \hookrightarrow \sigma_a$, by cases on the last rule in the derivation. Renaming is only necessary in the E-NEW case.

The most interesting subcase is that in which the E-PARAPP rule is the last rule in the derivation of both $\sigma \hookrightarrow \sigma_a$ and $\sigma \hookrightarrow \sigma_b$. Here, as Figure 5 illustrates, appealing to the induction hypothesis alone is not enough to complete the case, and Lemmas 1, 2, and 3 all play a role. For instance, suppose we have from the induction hypothesis that $\langle S_{a_1}; e_{a_1}\rangle$ steps to $\langle S_{c_1}; e_{c_1}\rangle$. To complete the case, we need to show that $\langle S_{a_1} \sqcup_S S_{a_2}; e_{a_1}\ e_{a_2}\rangle$ can take a step by the E-PARAPP rule. But for E-PARAPP to apply, we need to show that e_{a_1} can take a step beginning in the *larger* store of $S_{a_1} \sqcup_S S_{a_2}$. To do so, we can appeal to Lemma 1, which allows us to "frame on" the additional store S_{a_2} that has resulted from a parallel subcomputation. $\square$

We can readily restate Lemma 4 as Corollary 1:

Corollary 1 (Strong Local Confluence). *If $\sigma \hookrightarrow \sigma'$ and $\sigma \hookrightarrow \sigma''$, then there exist σ_c, i, j such that $\sigma' \hookrightarrow^i \sigma_c$ and $\sigma'' \hookrightarrow^j \sigma_c$ and $i \leq 1$ and $j \leq 1$.*

Proof. Choose $i = j = 1$. The proof follows immediately from Lemma 4. □

2.4.3. Confluence Lemmas and Determinism. With Lemma 4 in place, we can straightforwardly generalize its result to arbitrary numbers of steps by induction on the number of steps, as Lemmas 5, 6, and 7 show. [8]

Lemma 5 (Strong One-Sided Confluence). *If $\sigma \hookrightarrow \sigma'$ and $\sigma \hookrightarrow^m \sigma''$, where $1 \leq m$, then there exist σ_c, i, j such that $\sigma' \hookrightarrow^i \sigma_c$ and $\sigma'' \hookrightarrow^j \sigma_c$ and $i \leq m$ and $j \leq 1$.*

Proof sketch. By induction on m. In the base case of $m = 1$, the result is immediate from Corollary 1. □

Lemma 6 (Strong Confluence). *If $\sigma \hookrightarrow^n \sigma'$ and $\sigma \hookrightarrow^m \sigma''$, where $1 \leq n$ and $1 \leq m$, then there exist σ_c, i, j such that $\sigma' \hookrightarrow^i \sigma_c$ and $\sigma'' \hookrightarrow^j \sigma_c$ and $i \leq m$ and $j \leq n$.*

Proof sketch. By induction on n. In the base case of $n = 1$, the result is immediate from Lemma 5. □

Lemma 7 (Confluence). *If $\sigma \hookrightarrow^* \sigma'$ and $\sigma \hookrightarrow^* \sigma''$, then there exists σ_c such that $\sigma' \hookrightarrow^* \sigma_c$ and $\sigma'' \hookrightarrow^* \sigma_c$.*

Proof. Strong Confluence (Lemma 6) implies Confluence. □

Theorem 1 (Determinism). *If $\sigma \hookrightarrow^* \sigma'$ and $\sigma \hookrightarrow^* \sigma''$, and neither σ' nor σ'' can take a step except by* E-Refl *or* E-ReflErr*, then $\sigma' = \sigma''$.*

[8]Lemmas 5, 6, and 7 are nearly identical to the corresponding lemmas in the proof of determinism for Featherweight CnC given by Budimlić *et al.* [**9**]. We also reuse Budimlić *et al.*'s naming conventions for Lemmas 1 through 4, but they differ considerably in our setting due to the generality of LVars.

PROOF. We have from Lemma 7 that there exists σ_c such that $\sigma' \hookrightarrow^* \sigma_c$ and $\sigma'' \hookrightarrow^* \sigma_c$. Since σ' and σ'' can only step to themselves, we must have $\sigma' = \sigma_c$ and $\sigma'' = \sigma_c$, hence $\sigma' = \sigma''$. □

2.4.4. Discussion: Termination. Above we have followed Budimlić *et al.* [**9**] in treating *determinism* separately from the issue of *termination*. Yet one might legitimately be concerned that in λ_{LVar}, a configuration could have both an infinite reduction path and one that terminates with a value. Theorem 1 says that if two runs of a given λ_{LVar} program reach configurations where no more reductions are possible (except by reflexive rules), then they have reached the same configuration. Hence Theorem 1 handles the case of *deadlocks* already: a λ_{LVar} program can deadlock (*e.g.*, with a blocked `get`), but it will do so deterministically.

However, Theorem 1 has nothing to say about *livelocks*, in which a program reduces infinitely. It would be desirable to have a *consistent termination* property which would guarantee that if one run of a given λ_{LVar} program terminates with a non-**error** result, then every run will. We conjecture (but do not prove) that such a consistent termination property holds for λ_{LVar}. Such a property could be paired with Theorem 1 to guarantee that if one run of a given λ_{LVar} program terminates in a non-**error** configuration σ, then every run of that program terminates in σ. (The "non-**error** configuration" condition is necessary because it is possible to construct a λ_{LVar} program that can terminate in **error** on some runs and diverge on others. By contrast, our existing determinism theorem does not have to treat **error** specially.)

2.5. Safe, Limited Nondeterminism

In practice, a major problem with nondeterministic programs is that they can *silently* go wrong. Most parallel programming models are *unsafe* in this sense, but we may classify a nondeterministic language as *safe* if all occurrences of nondeterminism—that is, execution paths that would yield a wrong answer—are trapped and reported as errors. This notion of *safe nondeterminism* is analogous to the concept of type safety: type-safe programs can throw exceptions, but they will not "go wrong". We find that there are various extensions to a deterministic language that make it safely nondeterministic.[9] Here, we will look at one such extension: *exact but destructive observations*.

We begin by noting that when the state of an LVar has come to rest—when no more `put`s will occur—then its final value is a deterministic function of program inputs, and is therefore safe to read directly, rather than through a thresholded `get`. For instance, once no more elements will be added to the `l_acc` accumulator variable in the `bf_traverse` example of Figure 1, it is safe to read the exact, complete set contents.

The problem is determining automatically *when* an LVar has come to rest; usually, the programmer must determine this based on the control flow of the program. We may, however, provide a mechanism for the programmer to place their bet. *If* the value of an LVar is indeed at rest, then we do no harm to it by corrupting its state in such a way that further modification will lead to an error. We can accomplish this by adding an extra state, called *probation*, to D. The lattice defined by the relation $\sqsubseteq$ is extended thus:

$$\mathit{probation} \sqsubseteq \top$$

$$\forall d \in D.\ d \not\sqsubseteq \mathit{probation}$$

We then propose a new operation, `bump`, that takes a pointer to an LVar l, updates the store, setting l's state to *probation*, and returns a singleton set containing the

[9]For instance, while not recognized explicitly by the authors as such, a recent extension to CnC for memory management [**46**] incidentally fell into this category.

exact previous state of l, rather than a lower bound on that state. The idea is to ensure that, after a `bump`, any further operations on l will go awry: `put` operations will attempt to move the state of l to $\top$, resulting in **error**.

In the following example program, we use `bump` to perform an asynchronous sum reduction over a known number of inputs. In such a reduction, data dependencies alone determine when the reduction is complete, rather than control constructs such as parallel loops and barriers.

(Example 3)

```
let $cnt$ = new in
  let $sum$ = new in
    let par $p_1$ = (bump$_3$ $sum$; put $\{a\}$ $cnt$)
            $p_2$ = (bump$_4$ $sum$; put $\{b\}$ $cnt$)
            $p_3$ = (bump$_5$ $sum$; put $\{c\}$ $cnt$)
            $r$  = (get $cnt$ $\{a, b, c\}$; bump $sum$)
      in ... $r$ ...
```

In (Example 3), we use semicolon for sequential composition: $e_1; e_2$ is sugar for `let _ =` e_1 `in` e_2. We also assume a new syntactic sugar in the form of a `bump` operation that takes a pointer to an LVar representing a counter and increments it by one, with $\texttt{bump}_n\ l$ as an additional shorthand for n consecutive `bump`s to l. Meanwhile, the *cnt* LVar uses the power-set lattice of the set $\{a, b, c\}$ to track the completion of the p_1, p_2 and p_3 "threads".LK: If we had room, we could point out that the symbols a, b, and c are not important; any power-set lattice of a set of three elements would work here.

Before the call to `bump`, `get` *cnt* $\{a, b, c\}$ serves as a synchronization mechanism, ensuring that all increments are complete before the value is read. Three writers and one reader execute in parallel, and only when all writers complete does the reader return the sum, which in this case will be $3 + 4 + 5 = 12$.

The good news is that (Example 3) is deterministic; it will always return the same value in any execution. However, the `bump` primitive in general admits safe nondeterminism, meaning that, while all runs of the program will terminate with the same value *if* they terminate without error, some runs of the program may terminate in **error**, in spite of other runs completing successfully. To see how an error might occur, imagine an alternate version of (Example 3) in which `get` *cnt* $\{a, b, c\}$ is replaced by `get` *cnt* $\{a, b\}$. This version would have insufficient synchronization. The program could run correctly many times—if the `bump`s happen to complete before the `bump` operation executes—and yet step to **error** on the thousandth run. Yet, with safe nondeterminism, it is possible to catch and respond to this error, for example by re-running in a debug mode that is guaranteed to find a valid execution if it exists, or by using a *data-race detector* which will reproduce all races in the execution in question. In fact, the simplest form of error handling is to simply retry (or rerun from the last snapshot)—most data races make it into production only because they occur *rarely*. We have implemented a proof-of-concept interpreter and data-race detector for λ_{LVar} extended with `bump`, available in the LVars repository.

2.5.1. Desugaring `bump`. In this section we explain how the `bump` operation for LVar counters—as well as an underlying capability for generating unique IDs—can be implemented in λ_{LVar}.

Strictly speaking, if we directly used the atomic counter lattice of Figure 1(c) for the *sum* LVar in (Example 3), we would not be able to implement `bump`. Rather than use that lattice directly, then, we can simulate it using a power-set lattice over an arbitrary alphabet of symbols $\{s_1, s_2, s_3, \ldots\}$, ordered by subset inclusion. LVars whose states occupy such a lattice encode natural numbers using the cardinality of the subset.[10] With this encoding, incrementing a shared variable l requires `put` l $\{\alpha\}$, where $\alpha \in \{s_1, s_2, s_3, \ldots\}$ and α has not previously been used. Rather than requiring the programmer to be responsible for creating a unique α for each parallel contribution to the counter, though, we would like to be able to provide a language construct `unique` that, when evaluated, returns a singleton set containing a single unique element of the alphabet: $\{\alpha\}$. The expression `bump` l could then simply desugar to `put` l `unique`.

Fortunately, `unique` is implementable: well-known techniques exist for generating a unique (but schedule-invariant and deterministic) identifier for a given point in a parallel execution. One such technique is to reify the position of an operation inside a tree (or DAG) of parallel evaluations. The Cilk Plus parallel programming language refers to this notion as the *pedigree* of an operation and uses it to seed a deterministic parallel random number generator [**30**].

Figure **??** gives one possible set of rewrite rules for a transformation that uses the pedigree technique to desugar λ_{LVar} programs containing `unique`. Bearing some resemblance to a continuation-passing-style transformation [**15**], it creates a tree that tracks the dynamic evaluation of applications. We have implemented this transformation as a part of our interpreter for λ_{LVar} extended with `bump`. With `unique` in place, we can write programs like the following, in which two parallel computations

[10]Of course, just as with an encoding like Church numerals, this encoding would never be used by a realistic implementation.

increment the same counter:

```
let $sum$ = new in
  let par $p_1$ = (put $sum$ unique; put $sum$ unique)
          $p_2$ = (put $sum$ unique)
      in ...
```

In this case, the p_1 and p_2 "threads" will together increment the sum by three. Notice that consecutive increments performed by p_1 are not atomic.

LK: FIXME: don't include all this "unique" stuff.

2.6. Conclusion

As single-assignment languages and Kahn process networks demonstrate, monotonicity serves as the foundation of deterministic parallelism. Taking monotonicity as a starting point, our work generalizes single assignment to monotonic multiple assignment parameterized by a user-specified lattice. By combining monotonic writes with threshold reads, we get a shared-state parallel programming model that generalizes and unifies an entire class of monotonic languages suitable for asynchronous, data-driven applications. Our model is provably deterministic, and further provides a foundation for exploration of limited nondeterminism. Future work will improve upon our prototype implementation, formally establish the relationship between LVars and other deterministic parallel models, investigate consistent termination for λ_{LVar}, and prove the limited nondeterminism property of λ_{LVar} extended with `bump`.

CHAPTER 3

Quasi-Deterministic and Event-Driven Programming with LVars

TODO: Move Listings junk out of here.

3.1. Introduction

Flexible parallelism requires tasks to be scheduled dynamically, in response to the vagaries of an execution. But if the resulting schedule nondeterminism is *observable* within a program, it becomes much more difficult for programmers to discover and correct bugs by testing, let alone to reason about their code in the first place.

While much work has focused on identifying methods of deterministic parallel programming [**9, 29, 7, 27, 28, 52**], *guaranteed* determinism in real parallel programs remains a lofty and rarely achieved goal. It places stringent constraints on the programming model: concurrent tasks must communicate in restricted ways that prevent them from observing the effects of scheduling, a restriction that must be enforced at the language or runtime level.

The simplest strategy is to allow *no* communication, forcing concurrent tasks to produce values independently. Pure data-parallel languages follow this strategy [**43**], as do languages that force references to be either task-unique or immutable [**7**]. But some algorithms are more naturally or efficiently written using shared state or message passing. A variety of deterministic-by-construction models allow limited communication along these linesLK: ...so to speak, but they tend to be narrow in scope and permit communication through only a single data structure: for instance, FIFO queues in Kahn process networks [**27**] and StreamIt [**24**], or shared write-only tables in Intel Concurrent Collections [**9**].

Big-tent deterministic parallelism. Our goal is to create a broader, general-purpose deterministic-by-construction programming environment to increase the appeal and applicability of the method. We seek an approach that is not tied to a particular data structure and that supports familiar idioms from both functional and imperative programming styles. Our starting point is the idea of *monotonic* data structures, in which (1) information can only be added, never removed, and (2) the order in which information is added is not observable. A paradigmatic example is a set that supports insertion but not removal, but there are many others.

Our recently proposed *LVars* programming model [**36**] makes an initial foray into programming with monotonic data structures. LK: Since we call it the "LVars model" here, let's be consistent in calling it that, rather than "LVar model". In this model (which we review in Section 3.2), all shared data structures (called LVars) are monotonic, and the states that an LVar can take on form a *lattice*. Writes to an LVar must correspond to a *join* (least upper bound) in the lattice, which means that they monotonically increase the information in the LVar, and that they commute with one another. But commuting writes are not enough to guarantee determinism: if a read can observe whether or not a concurrent write has happened, then it can observe differences in scheduling. So in the LVars model, the answer to the question "has a write occurred?" (*i.e.*, is the LVar above a certain lattice value?) is always *yes*;

the reading thread will block until the LVar's contents reach a desired threshold. In a monotonic data structure, the absence of information is transient—another thread could add that information at any time—but the presence of information is forever.

The LVars model guarantees determinism, supports an unlimited variety of data structures (anything viewable as a lattice), and provides a familiar API, so it already achieves several of our goals. Unfortunately, it is not as general-purpose as one might hope.

Consider an 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 read and least-upper-bound write 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). But in the LVars model, asking whether a node is in a set means waiting until the node *is* in the set, and it is not clear how to lift this restriction while retaining determinism.

Monotonic data structures that can say "no". In this paper, we propose two additions to the LVars model that significantly extend its reach.

First, we add *event handlers*, a mechanism for attaching a callback function to an LVar that runs, asynchronously, whenever events arrive (in the form of monotonic updates to the LVar). Ordinary LVar reads encourage a synchronous, *pull* model of programming in which threads ask specific questions of an LVar, potentially blocking until the answer is "yes". Handlers, by contrast, support an asynchronous, *push* model of programming. Crucially, it is possible to check for *quiescence* of a handler, 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.

LK: I want to say something like: "If *all* writes happen through callbacks, we know that quiescence is really quiescence." (If we have some writes coming in otherwise, then quiescence is less helpful.)

Second, we add a primitive for *freezing* an LVar, which comes with the following tradeoff: once an LVar is frozen, any further writes that would change its value instead throw 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.[1]

Putting these features together, we can write a parallel graph traversal algorithm in the following simple fashion:

```
traverse :: Graph → NodeLabel → Par (Set NodeLabel)
traverse g startV = do
  seen ← newEmptySet
  putInSet seen startV
  let handle node = parMapM (putInSet seen) (nbrs g node)
```

[1]Our original work on LVars [**36**] included a brief sketch of a similar proposal for a "consume" operation on LVars, but did not study it in detail. Here, we include freezing in our model, prove quasi-determinism for it, and show how to program with it in conjunction with our other proposal, handlers.

```
  freezeSetAfter seen handle
```

This code, written using our Haskell implementation (described in Section 4.2),[2] discovers (in parallel) the set of nodes in a graph `g` reachable from a given node `startV`, and is guaranteed to produce a deterministic result. It works by creating a fresh `Set` LVar (corresponding to a lattice whose elements are sets, with set union as least upper bound), and seeding it with the starting node. The `freezeSetAfter` function combines the constructs proposed above. First, it installs the callback `handle` as a handler for the `seen` set, which will asynchronously put the neighbors of each visited node into the set, possibly triggering further callbacks, recursively. Second, when no further callbacks are ready to run—*i.e.*, when the `seen` set has reached a fixpoint—`freezeSetAfter` will freeze the set and return its exact value.

Quasi-determinism. Unfortunately, freezing does not commute with writes that change an LVar.[3] If a freeze is interleaved before such a write, the write will raise an exception; if it is interleaved afterwards, the program will proceed normally. It would appear that the price of negative information is the loss of determinism!

Fortunately, the loss is not total. Although LVar programs with freezing are not guaranteed to be deterministic, they do satisfy a related property that we call *quasi-determinism*: all executions that produce a final value produce the *same* final value. To put it another way, a quasi-deterministic program can be trusted to never change its answer due to nondeterminism; at worst, it might raise an exception on some runs. In our proposed model, this exception can in principle pinpoint the exact pair of freeze and write operations that are racing, greatly easing debugging.

[2]The `Par` type constructor is the monad in which LVar computations live.

[3]The same is true for quiescence detection; see Section 3.3.2.

Our general observation is that *pushing towards full-featured, general monotonic data structures leads to flirtation with nondeterminism*; perhaps the best way of ultimately getting deterministic outcomes is to traipse a small distance into nondeterminism, and make our way back. The identification of quasi-deterministic programs as a useful intermediate class is a contribution of this paper. That said, in many cases our freezing construct is only used as the very final step of a computation: after a global barrier, freezing is used to extract an answer. In this common case, we can guarantee determinism, since no writes can subsequently occur.

Contributions. The technical contributions of this paper are:

- We introduce *LVish*, a quasi-deterministic parallel programming model that extends LVars to incorporate freezing and event handlers (Section 3.3). In addition to our high-level design, we present a core calculus for LVish (Section 3.4), formalizing its semantics, and include a runnable version, implemented in PLT Redex (Section 3.4.7), for interactive experimentation.
- We give a proof of quasi-determinism for the LVish calculus (Section 3.5). The key lemma, Independence, gives a kind of *frame property* for LVish computations: very roughly, if a computation takes an LVar from state p to p', then it would take the same LVar from the state $p \sqcup p_F$ to $p' \sqcup p_F$. The Independence lemma captures the commutative effects of LVish computations.
- We describe a Haskell library for practical quasi-deterministic parallel programming based on LVish (Section 4.2). Our library comes with a number of monotonic data structures, including sets, maps, counters, and single-assignment variables. Further, it can be extended with new data structures, all of which can be used compositionally within the same program. Adding a new data structure typically involves porting an existing scalable (*e.g.*, *lock-free*) data structure to

Haskell, then wrapping it to expose a (quasi-)deterministic LVar interface. Our library exposes a monad that is *indexed* by a determinism level: fully deterministic or quasi-deterministic. Thus, the *static type* of an LVish computation reflects its guarantee, and in particular the freeze-last idiom allows freezing to be used safely with a fully-deterministic index.

- In Section 4.3, we evaluate our library with a case study: parallelizing control flow analysis. The case study begins with an existing implementation of k-CFA [**34**] written in a purely functional style. We show how this code can easily and safely be parallelized by adapting it to the LVish model—an adaptation that yields promising parallel speedup, and also turns out to have benefits even in the sequential case.

3.2. Background: the LVars Model

IVars [**3, 41, 9, 32**] are a well-known mechanism for deterministic parallel programming. An IVar is a *single-assignment* variable [**52**] with a blocking read semantics: an attempt to read an empty IVar will block until the IVar has been filled with a value. We recently proposed *LVars* [**36**] as a generalization of IVars: unlike IVars, which can only be written to once, LVars allow multiple writes, so long as those writes are monotonically increasing with respect to an application-specific lattice of states.

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:

$$\begin{array}{l} \texttt{let par } \_ = \texttt{put } lv\ 3 \\ \qquad\quad\ \_ = \texttt{put } lv\ 2 \\ \quad\ \texttt{in get } lv \end{array} \qquad \text{(Example 4)}$$

Here, `put` and `get` are operations that write and read LVars, respectively, and the expression

$$\texttt{let par } x_1 = e_1;\ x_2 = e_2;\ \ldots \texttt{ in } \mathit{body}$$

has *fork-join* semantics: it launches concurrent subcomputations $e_1, e_2, \ldots$ whose executions arbitrarily interleave, but must all complete before *body* runs. The `put` operation is defined in terms of the application-specific lattice of LVar states: it updates the LVar to the *least upper bound* of its current state and the new state being written.

If lv's lattice is the $\leq$ ordering on positive integers, as shown in Figure 1(a), then lv's state will always be $\max(3, 2) = 3$ by the time `get` lv runs, since the least upper bound of two positive integers n_1 and n_2 is $\max(n_1, n_2)$. Therefore Example 4 will deterministically evaluate to 3, regardless of the order in which the two `put` operations occurred.

On the other hand, if lv's lattice is that shown in Figure 1(b), in which the least upper bound of any two distinct positive integers is $\top$, then Example 4 will deterministically raise an exception, indicating that conflicting writes to lv have occurred. This exception is analogous to the "multiple `put`" error raised upon multiple writes to an IVar. Unlike with a traditional IVar, though, multiple writes of the *same* value (say, `put` lv 3 and `put` lv 3) will *not* raise an exception, because the least upper bound 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 reads. However, merely ensuring that writes to an LVar are monotonically increasing is not enough to ensure that programs behave deterministically. Consider again the lattice of Figure 1(a) for lv, but suppose we change Example 4 to allow the `get` operation to be interleaved with the two `put`s:

(Example 5)
$$\begin{aligned}&\texttt{let par}\ \_ = \texttt{put}\ lv\ 3\\ &\quad \_ = \texttt{put}\ lv\ 2\\ &\quad x = \texttt{get}\ lv\\ &\quad\ \texttt{in}\ x\end{aligned}$$

Since the two `put`s and the `get` can be scheduled in any order, Example 5 is non-deterministic: x might be either 2 or 3, depending on the order in which the LVar effects occur. Therefore, to maintain determinism, LVars put an extra restriction on the `get` operation. Rather than allowing `get` to observe the exact value of the LVar, it can only observe that the LVar has reached one of a specified set of *lower bound* states. This set of lower bounds, which we provide as an extra argument to `get`, is called a *threshold set* because the values in it form a “threshold” that the state of the LVar must cross before the call to `get` is allowed to unblock and return. When the threshold has been reached, `get` unblocks and returns *not* the exact value of the LVar, but instead, the (unique) element of the threshold set that has been reached or surpassed.

We can make Example 5 behave deterministically by passing a threshold set argument to `get`. LK: I cut out the separate code example for this to save space—it was only a couple characters different! For instance, suppose we choose the singleton set $\{3\}$ as the threshold set. Since lv's value can only increase with time, we know that once it is at least 3, it will remain at or above 3 forever; therefore the program will deterministically evaluate to 3. Had we chosen $\{2\}$ as the threshold set, the program would deterministically evaluate to 2; had we chosen $\{4\}$, it would deterministically block forever.

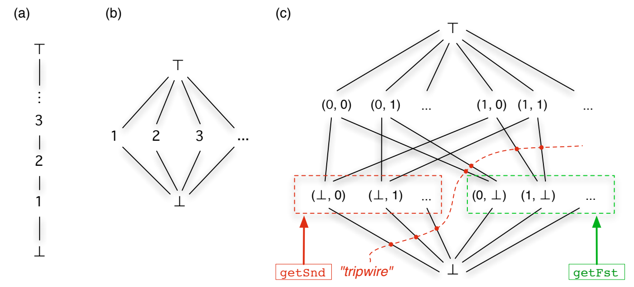

Figure 1. Example LVar lattices: (a) positive integers ordered by $\leq$; (b) IVar containing a positive integer; (c) pair of natural-number-valued IVars, annotated with example threshold sets that would correspond to a blocking read of the first or second element of the pair. Any state transition crossing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

As long as we only access LVars with `put` and (thresholded) `get`, we can arbitrarily share them between threads without introducing nondeterminism. That is, the `put` and `get` operations in a given program can happen in any order, without changing the value to which the program evaluates.

Incompatibility of threshold sets. While the LVar interface just described is deterministic, it is only useful for synchronization, not for communicating data: we must specify in advance the single answer we expect to be returned from the call to `get`. In general, though, threshold sets do not have to be singleton sets. For example, 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 `getFst` and `getSnd` operations for reading from the first and second entries of *lv*:

$$\texttt{getFst}\ p \overset{\triangle}{=} \texttt{get}\ p\ \{(m, \bot) \mid m \in \mathbb{N}\}$$

$$\texttt{getSnd}\ p \overset{\triangle}{=} \texttt{get}\ p\ \{(\bot, n) \mid n \in \mathbb{N}\}$$

This allows us to write programs like the following:

(Example 6)

$$\begin{array}{l}\texttt{let par}\ \_ = \texttt{put}\ lv\ (\bot, 4)\\ \quad \_ = \texttt{put}\ lv\ (3, \bot)\\ \quad x = \texttt{getSnd}\ lv\\ \quad \texttt{in}\ x\end{array}$$

In the call `getSnd` *lv*, the 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. (We formalize this incompatibility requirement in Section 3.4.5.)

In the case of Example 6, `getSnd` *lv* may unblock and return $(\bot, 4)$ 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.

The model versus reality. The use of explicit threshold sets in the above LVars model should be understood as a mathematical modeling technique, *not* an implementation approach or practical API. Our library (discussed in Section 4.2) provides an unsafe `getLV` operation to the authors of LVar data structure libraries, who can then make operations like `getFst` and `getSnd` available as a safe interface for application writers, implicitly baking in the particular threshold sets that make sense for a given data structure without ever explicitly constructing them.

To put it another way, operations on a data structure exposed as an LVar must have the *semantic effect* of a least upper bound for writes or a threshold for reads, but none of this need be visible to clients (or even written explicitly in the code). Any data structure API that provides such a semantics is guaranteed to provide deterministic concurrent communication.

3.3. LVish, Informally

As we explained in Section 3.1, while LVars offer a deterministic programming model that allows communication through a wide variety of data structures, they are not powerful enough to express common algorithmic patterns, like fixpoint computations, that require both positive and negative queries. In this section, we explain our extensions to the LVar model at a high level; Section 3.4 then formalizes them, while Section 4.2 shows how to implement them.

3.3.1. Asynchrony through Event Handlers.

Our first extension to LVars is the ability to do asynchronous, event-driven programming through event handlers. An *event* for an LVar can be represented by a lattice element; the event *occurs* when the LVar's current value reaches a point at or above that lattice element. An *event handler* ties together an LVar with a callback function that is asynchronously invoked whenever some events of interest occur. For example, if *lv* is an LVar whose lattice is that of Figure 1(a), the expression

(Example 7) $\texttt{addHandler}\ lv\ \{1,3,5,\dots\}\ (\lambda x.\, \texttt{put}\ lv\ x+1)$

registers a handler for lv that executes the callback function $\lambda x.\,\texttt{put}\ lv\ x+1$ for each odd number that lv is at or above. When Example 7 is finished evaluating, lv will contain the smallest even number that is at or above what its original value was. For instance, if lv originally contains 4, the callback function will be invoked twice, once with 1 as its argument and once with 3. These calls will respectively write $1+1=2$ and $3+1=4$ into lv; since both writes are ≤ 4, lv will remain 4. On the other hand, if lv originally contains 5, then the callback will run three times, with 1, 3, and 5 as its respective arguments, and with the latter of these calls writing $5+1=6$ into lv, leaving lv as 6.

In general, the second argument to `addHandler` is an arbitrary subset Q of the LVar's lattice, specifying which events should be handled. Like threshold sets, these *event sets* are a mathematical modeling tool only; they have no explicit existence in the implementation.

Event handlers in LVish are somewhat unusual in that they invoke their callback for *all* events in their event set Q that have taken place (*i.e.*, all values in Q less than or equal to the current LVar value), even if those events occurred prior to the handler being registered. To see why this semantics is necessary, consider the following, more subtle example:

(Example 8)
$$\begin{array}{l} \texttt{let par}\ \_ = \texttt{put}\ lv\ 0 \\ \quad \_ = \texttt{put}\ lv\ 1 \\ \quad \_ = \texttt{addHandler}\ lv\ \{0,1\}\ (\lambda x.\,\texttt{if}\ x=0\ \texttt{then put}\ lv\ 2) \\ \quad\ \texttt{in get}\ lv\ \{2\} \end{array}$$

Can Example 8 ever block? If a callback only executed for events that arrived after its handler was registered, or only for the largest event in its handler set that had occurred, then the example would be nondeterministic: it would block, or not, depending on how the handler registration was interleaved with the `puts`. By instead executing a handler's callback once for *each and every* element in its event set below or at the LVar's value, we guarantee quasi-determinism—and, for Example 8, guarantee the result of 2.

The power of event handlers is most evident for lattices that model collections, such as sets. For example, if we are working with lattices of sets of natural numbers, ordered by subset inclusion, then we can write the following function:

$$\texttt{forEach} = \lambda lv.\, \lambda f.\, \texttt{addHandler}\ lv\ \{\{0\}, \{1\}, \{2\}, \dots\}\ f$$

Unlike the usual `forEach` function found in functional programming languages, this function sets up a *permanent*, asynchronous flow of data from lv into the callback f. Functions like `forEach` can be used to set up complex, cyclic data-flow networks, as we will see in Section 4.3.

In writing `forEach`, we consider only the singleton sets to be events of interest, which means that if the value of lv is some set like $\{2, 3, 5\}$ then f will be executed once for each singleton subset ($\{2\}$, $\{3\}$, $\{5\}$)—that is, once for each element. In Section 4.2.2, we will see that this kind of handler set can be specified in a lattice-generic way, and in Section 4.2 we will see that it corresponds closely to our implementation strategy.

3.3.2. Quiescence through Handler Pools. Because event handlers are asynchronous, we need a separate mechanism to determine when they have reached a *quiescent* state, *i.e.*, when all callbacks for the events that have occurred have finished running. As we discussed in Section 3.1, detecting quiescence is crucial for implementing fixpoint computations. To build flexible data-flow networks, it is also helpful to be able to detect quiescence of multiple handlers simultaneously. Thus, our design includes *handler pools*, which are groups of event handlers whose collective quiescence can be tested.

The simplest way to use a handler pool is the following:

```
let h = newPool
  in addInPool h lv Q f;
     quiesce h
```

where *lv* is an LVar, Q is an event set, and f is a callback. Handler pools are created with the `newPool` function, and handlers are registered with `addInPool`, a variant of `addHandler` that takes a handler pool as an additional argument. Finally, `quiesce` blocks until a pool of handlers has reached a quiescent state.

Of course, whether or not a handler is quiescent is a non-monotonic property: we can move in and out of quiescence as more `put`s to an LVar occur, and even if all states at or below the current state have been handled, there is no way to know that more `put`s will not arrive to increase the state and trigger more callbacks. There is no risk to quasi-determinism, however, because `quiesce` does not yield any information about *which* events have been handled—any such questions must be asked through LVar functions like `get`. In practice, `quiesce` is almost always used together with freezing, which we explain next.

3.3.3. Freezing and the Freeze-After Pattern.

Our final addition to the LVar model is the ability to *freeze* an LVar, which forbids further changes to it, but in return allows its exact value to be read. We expose freezing through the function `freeze`, which takes an LVar as its sole argument, and returns the exact value of the LVar as its result. As we explained in Section 3.1, `put`s that would change the value of a frozen LVar instead raise an exception, and it is the potential for races between such `puts` and `freeze` that makes LVish quasi-deterministic, rather than fully deterministic.

Putting all the above pieces together, we arrive at a particularly common pattern of programming in LVish:

$$\begin{aligned}\texttt{freezeAfter} = \lambda lv.\, \lambda Q.\, \lambda f.\ & \texttt{let } h \ = \ \texttt{newPool} \\ & \texttt{in addInPool } h\ lv\ Q\ f; \\ & \quad \texttt{quiesce } h;\ \texttt{freeze } lv\end{aligned}$$

In this pattern, an event handler is registered for an LVar, subsequently quiesced, and then the LVar is frozen and its exact value is returned. A set-specific variant of this pattern, `freezeSetAfter`, was used in the graph traversal example in Section 3.1.

3.4. LVish, Formally

In this section, we present a core calculus for LVish—in particular, a quasi-deterministic, parallel, call-by-value λ-calculus extended with a store containing LVars. It extends the original LVar formalism to support event handlers and freezing. In comparison to the informal description given in the last two sections, we make two simplifications to keep the model lightweight:

- We parameterize the definition of the LVish calculus by a *single* application-specific lattice, representing the set of states that LVars in the calculus can take on. Therefore LVish is really a *family* of calculi, varying by choice of lattice. Multiple lattices can in principle be encoded using a sum construction, so this modeling choice is just to keep the presentation simple; in any case, our Haskell implementation supports multiple lattices natively.
- Rather than modeling the full ensemble of event handlers, handler pools, quiescence, and freezing as separate primitives, we instead formalize the "freeze-after"

pattern—which combined them—directly as a primitive. This greatly simplifies the calculus, while still capturing the essence of our programming model.

In this section we cover the most important aspects of the LVish core calculus. Complete details, including the proof of Lemma 8, are given in the companion technical report [**38**].

3.4.1. 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" in the rest of this paper. We also occasionally use D as a shorthand for the entire 4-tuple $(D, \sqsubseteq, \bot, \top)$ when its meaning is clear from the context.

3.4.2. Freezing. To model freezing, we need to generalize the notion of the state of an LVar to 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 is "freezable" except $\top$. Informally:

- Two unfrozen states are ordered according to the application-specific $\sqsubseteq$; that is, $(d, \mathsf{false}) \sqsubseteq_p (d', \mathsf{false})$ exactly when $d \sqsubseteq d'$.
- Two frozen states do not have an order, unless they are equal: $(d, \mathsf{true}) \sqsubseteq_p (d', \mathsf{true})$ exactly when $d = d'$.
- An unfrozen state (d, false) is less than or equal to a frozen state (d', 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, $(d, \mathsf{true}) \sqsubseteq_p (d', \mathsf{false})$ 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. Definition 6 and Lemma 8 formalize this notion.

Definition 6 (Lattice freezing). Suppose $(D, \sqsubseteq, \bot, \top)$ is a lattice. We define an operation $\text{Freeze}(D, \sqsubseteq, \bot, \top) \triangleq (D_p, \sqsubseteq_p, \bot_p, \top_p)$ as follows:

(1) D_p is a set defined as follows:

$$\begin{aligned} D_p \triangleq\ & \{(d, \mathit{frz}) \mid d \in (D - \{\top\}) \land \mathit{frz} \in \{\mathsf{true}, \mathsf{false}\}\} \\ & \cup \{(\top, \mathsf{false})\} \end{aligned}$$

(2) $\sqsubseteq_p \in \mathcal{P}(D_p \times D_p)$ is a binary relation defined as follows:

$$\begin{array}{lclcl} (d, \mathsf{false}) & \sqsubseteq_p & (d', \mathsf{false}) & \iff & d \sqsubseteq d' \\ (d, \mathsf{true}) & \sqsubseteq_p & (d', \mathsf{true}) & \iff & d = d' \\ (d, \mathsf{false}) & \sqsubseteq_p & (d', \mathsf{true}) & \iff & d \sqsubseteq d' \\ (d, \mathsf{true}) & \sqsubseteq_p & (d', \mathsf{false}) & \iff & d' = \top \end{array}$$

(3) $\bot_p \triangleq (\bot, \mathsf{false})$.

(4) $\top_p \stackrel{\triangle}{=} (\top, \mathsf{false})$.

Lemma 8 (Lattice structure). *If* $(D, \sqsubseteq, \bot, \top)$ *is a lattice then* $\text{Freeze}(D, \sqsubseteq, \bot, \top)$ *is as well.*

3.4.3. Stores. During the evaluation of 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 Loc, to its state, which is some pair (d, frz) from the set D_p.

Definition 7. A *store* is either a finite partial mapping $S : Loc \overset{\text{fin}}{\to} (D_p - \{\top_p\})$, or the distinguished element $\top_S$.

We use the notation $S[l \mapsto (d, \mathit{frz})]$ to denote extending S with a binding from l to (d, frz). If $l \in dom(S)$, then $S[l \mapsto (d, \mathit{frz})]$ 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 $[l_1 \mapsto (d_1, \mathit{frz}_1), l_2 \mapsto (d_2, \mathit{frz}_2), \ldots]$.

It is straightforward to lift the $\sqsubseteq_p$ operations defined on elements of D_p to the level of stores:

Definition 8. A store S is *less than or equal to* a store S' (written $S \sqsubseteq_S S'$) iff:

- $S' = \top_S$, or
- $dom(S) \subseteq dom(S')$ and for all $l \in dom(S)$, $S(l) \sqsubseteq_p S'(l)$.

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 (concretely defined in [**38**]). If, for example,

$$(d_1, \mathit{frz}_1) \sqcup_p (d_2, \mathit{frz}_2) = \top_p,$$

Given a lattice $(D, \sqsubseteq, \bot, \top)$ with elements $d \in D$:

$$
\begin{array}{rrl}
\text{configurations} & \sigma ::= & \langle S;\, e\rangle \mid \textsf{error} \\
\text{expressions} & e ::= & x \mid v \mid e\ e \mid \texttt{get}\ e\ e \mid \texttt{put}\ e\ e \mid \texttt{new} \mid \texttt{freeze}\ e \\
 & \mid & \texttt{freeze}\ e\ \texttt{after}\ e\ \texttt{with}\ e \\
 & \mid & \texttt{freeze}\ l\ \texttt{after}\ Q\ \texttt{with}\ \lambda x.\, e, \{e, \dots\}, H \\
\text{stores} & S ::= & [l_1 \mapsto p_1, \dots, l_n \mapsto p_n] \mid \top_S \\
\text{values} & v ::= & \texttt{()} \mid d \mid p \mid l \mid P \mid Q \mid \lambda x.\, e \\
\text{eval contexts} & E ::= & [\,] \mid E\ e \mid e\ E \mid \texttt{get}\ E\ e \mid \texttt{get}\ e\ E \mid \texttt{put}\ E\ e \\
 & \mid & \texttt{put}\ e\ E \mid \texttt{freeze}\ E \mid \texttt{freeze}\ E\ \texttt{after}\ e\ \texttt{with}\ e \\
 & \mid & \texttt{freeze}\ e\ \texttt{after}\ E\ \texttt{with}\ e \mid \texttt{freeze}\ e\ \texttt{after}\ e\ \texttt{with}\ E \\
 & \mid & \texttt{freeze}\ v\ \texttt{after}\ v\ \texttt{with}\ v, \{e \dots\ E\ e \dots\}, H
\end{array}
$$

$$
\begin{array}{rl}
\text{“handled” sets}\ H ::= & \{d_1, \dots, d_n\} \\
\text{threshold sets}\ P ::= & \{p_1, p_2, \dots\} \\
\text{event sets}\ Q ::= & \{d_1, d_2, \dots\}
\end{array}
\qquad
\begin{array}{rl}
\text{states}\ p ::= & (d, \mathit{frz}) \\
\text{status bits}\ \mathit{frz} ::= & \textsf{true} \mid \textsf{false}
\end{array}
$$

FIGURE 2. Syntax for λ_{LVish}.

then

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

A store containing a binding $l \mapsto (\top, \mathit{frz})$ can never arise during the execution of an LVish program, because, as we will see in Section 3.4.5, an attempted `put` that would take the value of l to $\top$ will raise an error.

3.4.4. The LVish Calculus. The syntax and operational semantics of the LVish calculus appear in Figures **??** and **??**, respectively. As we have noted, both the syntax and semantics are parameterized by the lattice $(D, \sqsubseteq, \bot, \top)$. The reduction relation

$\hookrightarrow$ is defined on *configurations* $\langle S; e\rangle$ comprising a store and an expression. The *error configuration*, written **error**, is a unique element added to the set of configurations, but we consider $\langle \top_S; e\rangle$ to be equal to **error** for all expressions e. The metavariable σ ranges over configurations.

LVish uses a reduction semantics based on evaluation contexts. The E-Eval-Ctxt rule is a standard context rule, allowing us to apply reductions within a context. The choice of context determines where evaluation can occur; in LVish, the order of evaluation is nondeterministic (that is, a given expression can generally reduce in various ways), 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 LVish, the arguments to `get`, `put`, `freeze`, and application expressions are *implicitly* evaluated concurrently.[4]

Arguments must be fully evaluated, however, before function application (β-reduction, modeled by the E-Beta rule) can occur. We can exploit this property to define `let par` as syntactic sugar:

$$\texttt{let par}\ x = e_1;\ y = e_2\ \texttt{in}\ e_3 \ \stackrel{\triangle}{=}\ ((\lambda x.\,(\lambda y.\,e_3))\ e_1)\ e_2$$

Because we do not reduce under λ-terms, we can sequentially compose e_1 before e_2 by writing $\texttt{let}\ \_ = e_1\ \texttt{in}\ e_2$, 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.

[4]This is in contrast to the original LVars formalism given in [**36**], which models parallelism with explicitly simultaneous reductions.LK: What's a polite way to say that the new way of doing things is much cleaner? :)

3.4.5. Semantics of `new`, `put`, and `get`. In LVish, the `new`, `put`, and `get` operations respectively create, write to, and read from LVars in the store:

- `new` (implemented by the E-New rule) extends the store with a binding for a new LVar whose initial state is $(\bot, \mathsf{false})$, and returns the location l of that LVar (*i.e.*, a pointer to the LVar).
- `put` (implemented by the E-Put and 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, 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 **error**.
- `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 *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 $p_2' \sqsubseteq_p p_1$, it would follow that $p_2 \sqcup_p p_2' = p_1 \neq \top_p$, which contradicts the requirement that P be pairwise incompatible.[5]

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 `put`s 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

[5]We stress that, although *incomp*(P) is given as a premise of the E-Get reduction rule (suggesting that it is checked at runtime), in our real 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; see Section 4.2.

is guaranteed to fault, because $p_1 \sqcup_p p_2 = \top_p$, 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 fault.

3.4.6. The `freeze − after − with` Primitive.

AJT: Note: most of the text that was here before is now explained in the previous section, so this just focuses on the semantic modeling.

The 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 "freeze-after" pattern described in Section 3.3.3. 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, LVish includes an auxiliary form,

$$\texttt{freeze}\ l\ \texttt{after}\ Q\ \texttt{with}\ \lambda x.\, e_0, \{e, \dots\}, H$$

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 $\{e, \dots\}$ 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 our 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 being frozen, any further `put`s that would have enabled additional callbacks will instead fault, raising **error** by way of the E-Put-Err rule. LK: N.B. This is the first place in the paper where the word "fault" appears, and it doesn't appear anywhere else aside from this section, so I thought I should explain what it means.

Therefore, freezing is a way of "betting" that once a collection of callbacks have completed, no further `put`s that change the LVar's value will occur. For a given run of a program, either all `put`s to an LVar arrive before it has been frozen, in which case the value returned by $\mathtt{freeze} - \mathtt{after} - \mathtt{with}$ is the lub of those values, or some `put` arrives after the LVar has been frozen, in which case the program will fault. And thus we have arrived at *quasi-determinism*: a program will always either evaluate to the same answer or it will fault.

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 `put`s either prior to $\texttt{freeze} - \texttt{after} - \texttt{with}$, or by the callback function within it (as will be the case for fixpoint computations). In practice, freezing is usually the very 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.

3.4.7. Modeling Lattice Parameterization in Redex. We have developed a runnable version of the LVish calculus[6] using the PLT Redex semantics engineering toolkit [**19**]. In the Redex of today, it is not possible to directly parameterize a language definition by a lattice.[7] Instead, taking advantage of Racket's syntactic abstraction capabilities, we define a Racket macro, `define-LVish-language`, that wraps a template implementing the lattice-agnostic semantics of Figure **??**, and takes the following arguments:

- a *name*, which becomes the *lang-name* passed to Redex's `define-language` form;
- a *"downset" operation*, a Racket-level procedure that takes a lattice element and returns the (finite) set of all lattice elements that are below that element (this operation is used to implement the semantics of $\texttt{freeze} - \texttt{after} - \texttt{with}$, in particular, to determine when the E-FREEZE-FINAL rule can fire);

[6]Available at `http://github.com/iu-parfunc/lvars`.

[7]See discussion at `http://lists.racket-lang.org/users/archive/2013-April/057075.html`.

- a *lub operation*, a Racket-level procedure that takes two lattice elements and returns a lattice element; and
- a (possibly infinite) set of *lattice elements* represented as Redex *patterns*.

Given these arguments, `define-LVish-language` generates a Redex model specialized to the application-specific lattice in question. For instance, to instantiate a model called `nat`, where the application-specific lattice is the natural numbers with `max` as the least upper bound, one writes:

```
(define-LVish-language nat downset-op max natural)
```

where `downset-op` is separately defined. Here, `downset-op` and `max` are Racket procedures. `natural` is a Redex pattern that has no meaning to Racket proper, but because `define-LVish-language` is a macro, `natural` is not evaluated until it is in the context of Redex.LK: This might be too much information, or a little confusing. It's a nice illustration of the power of macros, though. I'd welcome suggestions for how to word it differently.

3.5. Quasi-Determinism for LVish

Our proof of quasi-determinism for LVish formalizes the claim we make in Section 3.1: 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, we 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 the companion technical report [**38**].

3.5.1. Quasi-Determinism and Quasi-Confluence. Our main result, Theorem 2, 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 2 (Quasi-Determinism). *If $\sigma \hookrightarrow^* \sigma'$ and $\sigma \hookrightarrow^* \sigma''$, and neither σ' nor σ'' can take a step, then either:*

(1) *$\sigma' = \sigma''$ up to a permutation on locations π, or*
(2) *$\sigma' =$ **error** or $\sigma'' =$ **error**.*

Theorem 2 follows from a series of *quasi-confluence* lemmas. The most important of these, Strong Local Quasi-Confluence (Lemma 9), 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 9's result to multiple steps by induction on the number of steps, eventually building up to Theorem 2.

LEMMA 9 (Strong Local Quasi-Confluence). *If $\sigma \equiv \langle S;\, e\rangle \hookrightarrow \sigma_a$ and $\sigma \hookrightarrow \sigma_b$, then either:*

(1) *there exist π, i, j and σ_c such that $\sigma_a \hookrightarrow^i \sigma_c$ and $\sigma_b \hookrightarrow^j \pi(\sigma_c)$ and $i \leq 1$ and $j \leq 1$, or*
(2) *$\sigma_a \hookrightarrow$ **error** or $\sigma_b \hookrightarrow$ **error**.*

3.5.2. Independence. In order to show Lemma 9, we need a "frame property" for LVish that captures the idea that independent effects commute with each other. Lemma 10, 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 $S \sqcup_S S''$: first, it will update the store to $S' \sqcup_S S''$; second, it will step to e' as it did before.LK: Aaron: Do you think it's fair to claim that these respectively boil down to the Safety Monotonicity and Frame Property claims from the "Local Action" paper? Here $S \sqcup_S S''$ 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 10 (Independence). *If* $\langle S; e\rangle \hookrightarrow \langle S'; e'\rangle$ *(where* $\langle S'; e'\rangle \neq$ **error***), then we have that:*

$\langle S \sqcup_S S''; e\rangle \hookrightarrow \langle S' \sqcup_S S''; e'\rangle$,

where S'' *is any store meeting the following conditions:*

- S'' *is non-conflicting with* $\langle S; e\rangle \hookrightarrow \langle S'; e'\rangle$,
- $S' \sqcup_S S'' =_{frz} S$, *and*
- $S' \sqcup_S S'' \neq \top_S$.

Lemma 10 requires as a precondition that the stores $S' \sqcup_S S''$ and 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 S'' must be *non-conflicting* with the original transition from $\langle S; e\rangle$ to $\langle S'; e'\rangle$, meaning that locations in S'' cannot share names with locations newly allocated during the transition; this rules out location name conflicts caused by allocation.

Definition 9. Two stores S and S' are *equal in status* (written $S =_{frz} S'$) iff for all $l \in (dom(S) \cap dom(S'))$,
if $S(l) = (d, frz)$ and $S'(l) = (d', frz')$, then $frz = frz'$.

Definition 10. A store S'' is *non-conflicting* with the transition $\langle S; e\rangle \hookrightarrow \langle S'; e'\rangle$ iff $(dom(S') - dom(S)) \cap dom(S'') = \emptyset$.

Given a lattice $(D, \sqsubseteq, \bot, \top)$ with elements $d \in D$:

$incomp(P) \overset{\triangle}{=} \forall\, p_1, p_2 \in P.\ (p_1 \neq p_2 \implies p_1 \sqcup_p p_2 = \top_p)$ $\boxed{\sigma \hookrightarrow \sigma'}$

E-Eval-Ctxt

$$\frac{\langle S;\, e\rangle \hookrightarrow \langle S';\, e'\rangle}{\langle S;\, E[e]\rangle \hookrightarrow \langle S';\, E\big[e'\big]\rangle}$$

E-Beta

$$\frac{}{\langle S;\, (\lambda x.\, e)\ v\rangle \hookrightarrow \langle S;\, e[x := v]\rangle}$$

E-New

$$\frac{}{\langle S;\, \texttt{new}\rangle \hookrightarrow \langle S[l \mapsto (\bot, \mathsf{false})];\, l\rangle}\ (l \notin dom(S))$$

E-Put

$$\frac{S(l) = p_1 \qquad p_2 = p_1 \sqcup_p (d_2, \mathsf{false}) \qquad p_2 \neq \top_p}{\langle S;\, \texttt{put}\ l\ d_2\rangle \hookrightarrow \langle S[l \mapsto p_2];\, \texttt{()}\rangle}$$

E-Put-Err

$$\frac{S(l) = p_1 \qquad p_1 \sqcup_p (d_2, \mathsf{false}) = \top_p}{\langle S;\, \texttt{put}\ l\ d_2\rangle \hookrightarrow \textbf{error}}$$

E-Get

$$\frac{S(l) = p_1 \qquad incomp(P) \qquad p_2 \in P \qquad p_2 \sqsubseteq_p p_1}{\langle S;\, \texttt{get}\ l\ P\rangle \hookrightarrow \langle S;\, p_2\rangle}$$

E-Freeze-Init

$$\frac{}{\langle S;\, \texttt{freeze}\ l\ \texttt{after}\ Q\ \texttt{with}\ \lambda x.\, e\rangle \hookrightarrow \langle S;\, \texttt{freeze}\ l\ \texttt{after}\ Q\ \texttt{with}\ \lambda x.\, e, \{\}, \{\}\rangle}$$

E-Spawn-Handler

$$\frac{S(l) = (d_1, frz_1) \qquad d_2 \sqsubseteq d_1 \qquad d_2 \notin H \qquad d_2 \in Q}{\langle S;\, \texttt{freeze}\ l\ \texttt{after}\ Q\ \texttt{with}\ \lambda x.\, e_0, \{e, \ldots\}, H\rangle \hookrightarrow \langle S;\, \texttt{freeze}\ l\ \texttt{after}\ Q\ \texttt{with}\ \lambda x.\, e_0, \{e_0[x := d_2], e, \ldots\}}$$

E-Freeze-Final

$$\frac{S(l) = (d_1, frz_1) \qquad \forall d_2\ .\ (d_2 \sqsubseteq d_1 \land d_2 \in Q \Rightarrow d_2 \in H)}{\langle S;\, \texttt{freeze}\ l\ \texttt{after}\ Q\ \texttt{with}\ v, \{v \ldots\}, H\rangle \hookrightarrow \langle S[l \mapsto (d_1, \mathsf{true})];\, d_1\rangle}$$

E-Freeze-Simple

$$\frac{S(l) = (d_1, frz_1)}{\langle S;\, \texttt{freeze}\ l\rangle \hookrightarrow \langle S[l \mapsto (d_1, \mathsf{true})];\, d_1\rangle}$$

FIGURE 3. An operational semantics for λ_{LVish}.

CHAPTER 4

The LVish Library: Implementation and Evaluation

LK: What follows is from the FHPC paper and needs to be edited or removed.

4.1. Prototype Implementation and Evaluation

We have implemented a prototype LVars library based on the *monad-par* Haskell library, which provides the `Par` monad [**32**]. Our library, together with example programs and preliminary benchmarking results, is available in in the LVars repository. The relationship to λ_{LVar} is somewhat loose: for instance, while evaluation in λ_{LVar} is always strict, our library allows lazy, pure Haskell computations along with strict, parallel monadic computations.

A layered implementation. Use of our LVars library typically involves two parties: first, the data structure author who uses the library directly and provides an implementation of a specific monotonic data structure (*e.g.*, a monotonically growing hashmap), and, second, the application writer who uses that data structure. Only the application writer receives a determinism guarantee; it is the data structure author's obligation to ensure that the states of their data structure form a lattice and that it is only accessible via the equivalent of `put` and `get`.

The data structure author uses an interface provided by our LVars library, which provides core runtime functionality: thread scheduling and tracking of threads blocked on `get` operations. Concretely, the data structure author imports our library and reexports a limited interface specific to their data structure (*e.g.*, for sets, `putInSet` and

```
-- l_acc is an LVar "output parameter":
bf_traverse :: ISet NodeLabel → Graph →
               NodeLabel → Par ()
bf_traverse l_acc g startV =
  do putInSet l_acc startV
     loop {} {startV}
 where loop seen nu =
         if nu == {}
         then return ()
         else do
           let seen' = union seen nu
           allNbr  ← parMap (nbrs g) nu
           allNbr' ← parFold union allNbr
           let nu' = difference allNbr' seen'
           -- Add to the accumulator:
           parMapM (putInSet l_acc) nu'
           loop seen' nu'
-- The function 'analyze' is applied to everything
-- that is added to the set 'analyzeSet':
go = do analyzedSet ← newSetWith analyze
        res ← bf_traverse analyzedSet profiles profile0
```

FIGURE 1. An example Haskell program, written using our LVars library, that maps a computation over a connected component using a monotonically growing set variable. The code is written in a strict style, using the `Par` monad for parallelism. The use of the set variable enables modularity and safe pipelining. Consumers can safely asynchronously execute work items put into `analyzedSet`.

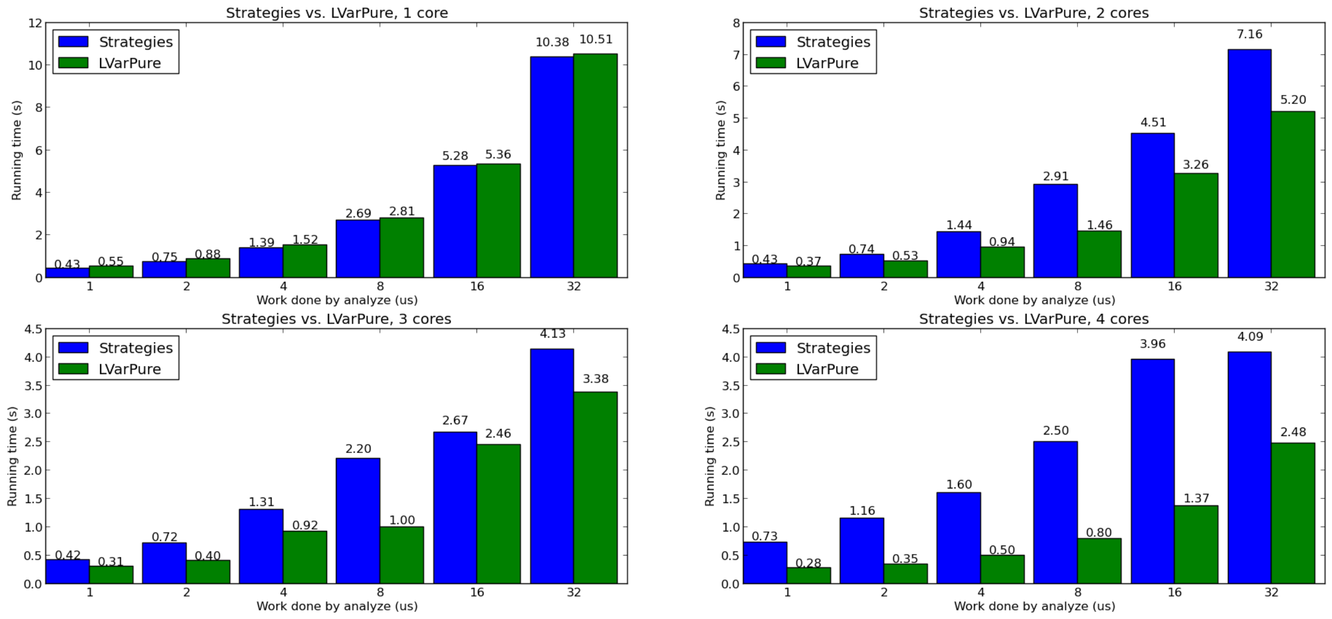

FIGURE 2. Running time comparison of Strategies-based and LVar-based implementations of `bf_traverse`, running on 1, 2, 3, and 4 cores on an Intel Xeon Core i5 3.1GHz (smaller is better).

`waitSetSizeThreshold`). In fact, our library provides three different runtime interfaces for the data structure author to choose among. These "layered" interfaces provide the data structure author with a shifting trade-off between the *ease* of meeting their proof obligation, and *attainable performance*:

(1) **Pure LVars**: Here, each LVar is a single mutable container (an `IORef`) containing a pure value. This requires only that a library writer select a purely functional data structure and provide a `join`[1] function for it and a threshold predicate for each `get` operation. These pure functions are easiest to validate, for example, using the popular QuickCheck [?] tool.

[1]Type class `Algebra.Lattice.JoinSemiLattice`.

(2) **IO LVars**: Pure data structures in mutable containers cannot always provide the best performance for concurrent data structures. Thus we provide a more effectful interface. With it, data structures are represented in terms of arbitrary mutable state; performing a `put` requires an *update action* (`IO`) and a `get` requires an effectful polling function that will be run after any `put` to the LVar, to determine if the `get` can unblock.

(3) **Scalable LVars**: Polling each blocked `get` upon *any* `put` is not very precise. If the data structure author takes on yet more responsibility, they can use our third interface to reduce contention by managing the storage of blocked computations and threshold functions *themselves*. For example, a concurrent set might store a waitlist of blocked continuations on a per-element basis, rather than using one waitlist for the entire LVar, as in layers (1) and (2).

The support provided to the data structure author *declines* with each of these layers, with the final option providing only parallel scheduling, and little help with defining the specific LVar data structure. But this progressive cost/benefit tradeoff can be beneficial for prototyping and then refining data structure implementations.

Revisiting our breadth-first traversal example. In Section **??**, we proposed using LVars to implement a program that performs a breadth-first traversal of a connected component of a graph, mapping a function over each node in the component. Figure 1 gives a version of this program implemented using our LVars library. It performs a breadth-first traversal of the `profiles` graph with effectful `put` operations on a shared set variable. This variable, `analyzedSet`, has to be modified multiple times by `putInSet` and thus cannot be an IVar.[2] The callback function `analyze` is "baked into" `analyzedSet` and may run as soon as new elements are inserted. Our implementation

[2]Indeed, there are subtle problems with encoding a set even as a linked structure of IVars. For example, if it is represented as a tree, who writes the root?

uses the "Pure LVars" runtime layer described above: `analyzedSet` is nothing more than a tree-based data structure (`Data.Set`) stored in a mutable location.

Preliminary benchmarking results. We compared the performance of the LVar-based implementation of `bf_traverse` against the version in Figure 1, which we ran using the

`Control.Parallel.Strategies` library [**31**], version 3.2.0.3. (Despite being a simple algorithm, even breadth-first search by itself is considered a useful benchmark; in fact, the well-known "Graph 500" [**?**] benchmark is exactly breadth-first search.)

We evaluated the Strategies and LVar versions of `bf_traverse` by running both on a local random directed graph of 40,000 nodes and 320,000 edges (and therefore an average degree of 8), simulating the `analyze` function by doing a specified amount of work for each node, which we varied from 1 to 32 microseconds. Figure 2 shows the results of our evaluation on 1, 2, 3, and 4 cores. Although both the Strategies and LVar versions enjoyed a speedup as we added parallel resources, the LVar version scaled particularly well. A subtler, but interesting point is that, in the Strategies version, it took an average of 64.64 milliseconds for the first invocation of `analyze` to begin running after the program began, whereas in the LVar version, it took an average of only 0.18 milliseconds, indicating that the LVar version allows work to be pipelined.

RRN: Report time-to-first-f results in prose I suppose....

LK: What follows is from the POPL paper and needs to be edited.

4.2. Implementation

We have constructed a prototype implementation of LVish as a monadic library in Haskell, which is available at `http://hackage.haskell.org/package/lvish`

AJT: Or would we rather link to github?Our library adopts the basic approach of the `Par` monad [**32**], enabling us to employ our own notion of lightweight, library-level threads with a custom scheduler. It supports the programming model laid out in Section 3.3 in full, including explicit handler pools. It differs from our formal model in following Haskell's by-need evaluation strategy, which also means that concurrency in the library is *explicitly marked*, either through uses of a `fork` function or through asynchronous callbacks, which run in their own lightweight thread.

Implementing LVish as a Haskell library makes it possible to provide compile-time guarantees about determinism and quasi-determinism, because programs written using our library run in our `Par` monad and can therefore only perform LVish-sanctioned side effects. We take advantage of this fact by indexing `Par` computations with a phantom type that indicates their *determinism level*:

```
data Determinism = Det | QuasiDet
```

The `Par` type constructor has the following kind:[3]

```
Par :: Determinism → ∗ → ∗
```

together with the following suite of `run` functions:

```
runPar   :: Par Det a → a
runParIO :: Par lvl a → IO a
runParThenFreeze :: DeepFrz a ⇒ Par Det a → FrzType a
```

[3]We are here using the `DataKinds` extension to Haskell to treat `Determinism` as a kind. In the full implementation, we include a second phantom type parameter to ensure that LVars cannot be used in multiple runs of the `Par` monad, in a manner analogous to how the `ST` monad prevents an `STRef` from being returned from `runST`.

The public library API ensures that if code uses `freeze`, it is marked as `QuasiDet`; thus, code that types as `Det` is guaranteed to be fully deterministic. While LVish code with an arbitrary determinism level `lvl` can be executed in the `IO` monad using `runParIO`, only `Det` code can be executed as if it were pure, since it is guaranteed to be free of visible side effects of nondeterminism. In the common case that `freeze` is only needed at the end of an otherwise-deterministic computation, `runParThenFreeze` runs the computation to completion, and then freezes the returned LVar, returning its exact value—and is guaranteed to be deterministic.[4]

AJT: We don't actually prove that freeze-free code is deterministic, but it should follow pretty easily from the proof of quasi-determinism. Perhaps we should claim it in the proof section? LK: Well, to do this "right" we would have to not just prove that freeze-free code is deterministic, but we'd have to prove that freeze-happens-last code is deterministic...

4.2.1. The Big Picture. We envision two parties interacting with our library. First, there are data structure authors, who use the library directly to implement a specific monotonic data structure (*e.g.*, a monotonically growing finite map). Second, there are application writers, who are clients of these data structures. Only the application writers receive a (quasi-)determinism guarantee; an author of a data structure is responsible for ensuring that the states their data structure can take on correspond to the elements of a lattice, and that the exposed interface to it corresponds to some use of `put`, `get`, `freeze`, and event handlers.

Thus, our library is focused primarily on *lattice-generic* infrastructure: the `Par` monad itself, a thread scheduler, support for blocking and signaling threads, handler pools,

[4]The `DeepFrz` typeclass is used to perform freezing of nested LVars, producing values of frozen type (as given by the `FrzType` type function).

and event handlers. Since this infrastructure is unsafe (does not guarantee quasi-determinism), only data structure authors should import it, subsequently exporting a *limited* interface specific to their data structure. For finite maps, for instance, this interface might include key/value insertion, lookup, event handlers and pools, and freezing—along with higher-level abstractions built on top of these.

For this approach to scale well with available parallel resources, it is essential that the data structures themselves support efficient parallel access; a finite map that was simply protected by a global lock would force all parallel threads to sequentialize their access. Thus, we expect data structure authors to draw from the extensive literature on scalable parallel data structures, employing techniques like fine-grained locking and lock-free data structures [**26**]. Data structures that fit into the LVish model have a special advantage: because all updates must commute, it may be possible to avoid the expensive synchronization which *must* be used for non-commutative operations [**4**]. And in any case, monotonic data structures are usually much simpler to represent and implement than general ones.

4.2.2. Two Key Ideas.

Leveraging atoms. Monotonic data structures acquire "pieces of information" over time. In a lattice, the smallest such pieces are called the *atoms* of the lattice: they are elements not equal to $\bot$, but for which the only smaller element is $\bot$. Lattices for which every element is the lub of some set of atoms are called *atomistic*, and in practice most application-specific lattices used by LVish programs have this property—especially those whose elements represent collections.

In general, the LVish primitives allow arbitrarily large queries and updates to an LVar. But for an atomistic lattice, the corresponding data structure usually exposes operations that work at the atom level, semantically limiting `puts` to atoms, `gets` to threshold sets of atoms, and event sets to sets of atoms. For example, the lattice of finite maps is atomistic, with atoms consisting of all singleton maps (*i.e.*, all key/value pairs). The interface to a finite map usually works at the atom level, allowing addition of a new key/value pair, querying of a single key, or traversals (which we model as handlers) that walk over one key/value pair at a time.

Our implementation is designed to facilitate good performance for atomistic lattices by associating LVars with a set of *deltas* (changes), as well as a lattice. For atomistic lattices, the deltas are essentially just the atoms—for a set lattice, a delta is an element; for a map, a key/value pair. Deltas provide a compact way to represent a change to the lattice, allowing us to easily and efficiently communicate such changes between `put`s and `get`s/handlers.

Leveraging idempotence. While we have emphasized the commutativity of least upper bounds, they also provide another important property: *idempotence*, meaning that $d \sqcup d = d$ for any element d. In LVish terms, repeated `put`s or `freeze`s have no effect, and since these are the only way to modify the store, the result is that $e; e$ behaves the same as e for any LVish expression e. Idempotence has already been recognized as a useful property for work-stealing scheduling [**33**]: if the scheduler is allowed to occasionally duplicate work, it is possible to substantially save on synchronization costs. Since LVish computations are guaranteed to be idempotent, we could use such a scheduler (for now we use the standard Chase-Lev deque [**13**]). But idempotence also helps us deal with races between `put` and `get`/`addHandler`, as we explain below.

4.2.3. Representation Choices.

Our library uses the following generic representation for LVars:

```
data LVar a d =
     LVar { state :: a,  status :: IORef (Status d) }
```

where the type parameter `a` is the (mutable) data structure representing the lattice, and `d` is the type of deltas for the lattice.[5] The `status` field is a mutable reference that represents the status bit:

```
data Status d = Frozen | Active (B.Bag (Listener d))
```

The status bit of an LVar is tied together with a bag of waiting *listeners*, which include blocked `get`s and handlers; once the LVar is frozen, there can be no further events to listen for.[6] The bag module (imported as `B`) supports atomic insertion and removal, and *concurrent* traversal:

```
put     :: Bag a → a → IO (Token a)
remove  :: Token a → IO ()
foreach :: Bag a → (a → Token a → IO ()) → IO ()
```

Removal of elements is done via abstract *tokens*, which are acquired by insertion or traversal. Updates may occur concurrently with a traversal, but are not guaranteed to be visible to it.

A listener for an LVar is a pair of callbacks, one called when the LVar's lattice value changes, and the other when the LVar is frozen:

```
data Listener d = Listener {
  onUpd :: d → Token (Listener d) → SchedQ → IO (),
  onFrz ::      Token (Listener d) → SchedQ → IO () }
```

The listener is given access to its own token in the listener bag, which it can use to deregister from future events (useful for a `get` whose threshold has been passed).

[5]For non-atomistic lattices, we take `a` and `d` to be the same type.

[6]In particular, with one atomic update of the flag we both mark the LVar as frozen and allow the bag to be garbage-collected.

It is also given access to the CPU-local scheduler queue, which it can use to spawn threads.

4.2.4. The Core Implementation. Internally, the `Par` monad represents computations in continuation-passing style, in terms of their interpretation in the `IO` monad:

```
type ClosedPar = SchedQ → IO ()
type ParCont a = a → ClosedPar
mkPar :: (ParCont a → ClosedPar) → Par lvl a
```

The `ClosedPar` type represents ready-to-run `Par` computations, which are given direct access to the CPU-local scheduler queue. Rather than returning a final result, a completed `ClosedPar` computation must call the scheduler, `sched`, on the queue. A `Par` computation, on the other hand, completes by passing its intended result to its continuation—yielding a `ClosedPar` computation.

Figure 3 gives the implementation for three core lattice-generic functions: `getLV`, `putLV`, and `freezeLV`, which we explain next.

Threshold reading. The `getLV` function assists data structure authors in writing operations with `get` semantics. In addition to an LVar, it takes two *threshold functions*, one for global state and one for deltas. The *global threshold* `gThresh` is used to initially check whether the LVar is above some lattice value(s) by global inspection; the extra boolean argument gives the frozen status of the LVar. The *delta threshold* `dThresh` checks whether a particular update takes the state of the LVar above some lattice state(s). Both functions return `Just r` if the threshold has been passed, where `r` is the result of the read. To continue our running example of finite maps with key/value pair deltas, we can use `getLV` internally to build the following `getKey` function that is exposed to application writers:

```
-- Wait for the map to contain a key; return its value
```

```
getLV :: (LVar a d) → (a → Bool → IO (Maybe b))
                    → (d → IO (Maybe b)) → Par lvl b
getLV (LVar{state, status}) gThresh dThresh =
  mkPar $λk q →
    let onUpd d = unblockWhen (dThresh d)
        onFrz   = unblockWhen (gThresh state True)
        unblockWhen thresh tok q = do
          tripped ← thresh
          whenJust tripped $ λb → do
            B.remove tok
            Sched.pushWork q (k b)
    in do
      curStat ← readIORef status
      case curStat of
        Frozen → do   -- no further deltas can arrive!
          tripped ← gThresh state True
          case tripped of
            Just b   → exec (k b) q
            Nothing → sched q
        Active ls → do
          tok ← B.put ls (Listener onUpd onFrz)
          frz ← isFrozen status -- must recheck after
                                -- enrolling listener
          tripped ← gThresh state frz
          case tripped of
            Just b   → do
              B.remove tok  -- remove the listener
              k b q         -- execute our continuation
            Nothing → sched q
```

```
putLV :: LVar a d → (a → IO (Maybe d)) → Par lvl ()
putLV (LVar{state, status}) doPut = mkPar $ λk q → do
  Sched.mark q  -- publish our intent to modify the LVar
```

```
getKey key mapLV = getLV mapLV gThresh dThresh where
  gThresh m frozen = lookup key m
  dThresh (k,v) | k == key  = return (Just v)
                | otherwise = return Nothing
```

where `lookup` imperatively looks up a key in the underlying map.

The challenge in implementing `getLV` is the possibility that a *concurrent* `put` will push the LVar over the threshold. To cope with such races, `getLV` employs a somewhat pessimistic strategy: before doing anything else, it enrolls a listener on the LVar that will be triggered on any subsequent updates. If an update passes the delta threshold, the listener is removed, and the continuation of the `get` is invoked, with the result, in a new lightweight thread. *After* enrolling the listener, `getLV` checks the *global* threshold, in case the LVar is already above the threshold. If it is, the listener is removed, and the continuation is launched immediately; otherwise, `getLV` invokes the scheduler, effectively treating its continuation as a blocked thread.

By doing the global check only after enrolling a listener, `getLV` is sure not to miss any threshold-passing updates. It does *not* need to synchronize between the delta and global thresholds: if the threshold is passed just as `getLV` runs, it might launch the continuation twice (once via the global check, once via delta), but by idempotence this does no harm. This is a performance tradeoff: we avoid imposing extra synchronization on *all* uses of `getLV` at the cost of some duplicated work in a rare case. We can easily provide a second version of `getLV` that makes the alternative tradeoff, but as we will see below, idempotence plays an *essential* role in the analogous situation for handlers.

Putting and freezing. On the other hand, we have the `putLV` function, used to build operations with `put` semantics. It takes an LVar and an *update function* `doPut` that performs the `put` on the underlying data structure, returning a delta if the `put` actually changed the data structure. If there is such a delta, `putLV` subsequently invokes all currently-enrolled listeners on it.

The implementation of `putLV` is complicated by another race, this time with freezing. If the `put` is nontrivial (*i.e.*, it changes the value of the LVar), the race can be resolved in two ways. Either the freeze takes effect first, in which case the `put` must fault, or else the `put` takes effect first, in which case both succeed. Unfortunately, we have no means to both check the frozen status *and* attempt an update in a single atomic step.[7]

Our basic approach is to ask forgiveness, rather than permission: we eagerly perform the `put`, and only afterwards check whether the LVar is frozen. Intuitively, this is allowed because *if* the LVar is frozen, the `Par` computation is going to terminate with an exception—so the effect of the `put` cannot be observed!

Unfortunately, it is not enough to *just* check the status bit for frozenness afterward, for a rather subtle reason: suppose the `put` is executing concurrently with a `get` which it causes to unblock, and that the `get`ting thread subsequently freezes the LVar. In this case, we *must* treat the `freeze` as if it happened after the `put`, because the `freeze` could not have occurred had it not been for the `put`. But, by the time `putLV` reads the status bit, it may already be set, which naively would cause `putLV` to fault.

To guarantee that such confusion cannot occur, we add a *marked* bit to each CPU scheduler state. The bit is set (using `Sched.mark`) prior to a `put` being performed,

[7]While we could require the underlying data structure to support such transactions, doing so would preclude the use of existing lock-free data structures, which tend to use a single-word compare-and-set operation to perform atomic updates. Lock-free data structures routinely outperform transaction-based data structures [**21**].

and cleared (using `Sched.clear`) only *after* `putLV` has subsequently checked the frozen status. On the other hand, `freezeLV` waits until it has observed a (transient!) clear mark bit on every CPU (using `Sched.awaitClear`) before actually freezing the LVar. This guarantees that any `puts` that *caused* the freeze to take place check the frozen status *before* the freeze takes place; additional `puts` that arrive concurrently may, of course, set a mark bit again after `freezeLV` has observed a clear status.

The proposed approach requires no barriers or synchronization instructions (assuming that the `put` on the underlying data structure acts as a memory barrier). Since the mark bits are per-CPU flags, they can generally be held in a core-local cache line in exclusive mode—meaning that marking and clearing them is extremely cheap. The only time that the busy flags can create cross-core communication is during `freezeLV`, which should only occur once per LVar computation.

One final point: unlike `getLV` and `putLV`, which are polymorphic in their determinism level, `freezeLV` is statically `QuasiDet`.

Handlers, pools and quiescence. Given the above infrastructure, the implementation of handlers is relatively straightforward. We represent handler pools as follows:

```
data HandlerPool = HandlerPool {
  numCallbacks :: Counter,  blocked :: B.Bag ClosedPar }
```

where `Counter` is a simple counter supporting atomic increment, decrement, and checks for equality with zero.[8] We use the counter to track the number of currently-executing callbacks, which we can use to implement `quiesce`. A handler pool also keeps a bag of threads that are blocked waiting for the pool to reach a quiescent state.

[8]One can use a high-performance *scalable non-zero indicator* [**18**] to implement `Counter`, but we have not yet done so.

We create a pool using `newPool` (of type `Par lvl HandlerPool`), and implement quiescence testing as follows:

```
quiesce :: HandlerPool → Par lvl ()
quiesce hp@(HandlerPool cnt bag) = mkPar $ λk q → do
  tok ← B.put bag (k ())
  quiescent ← poll cnt
  if quiescent then do B.remove tok; k () q
               else sched q
```

where the `poll` function indicates whether `cnt` is (transiently) zero. Note that we are following the same listener-enrollment strategy as in `getLV`, but with `blocked` acting as the bag of listeners.

Finally, `addHandler` has the following interface:

```
addHandler ::
     Maybe HandlerPool               -- Pool to enroll in
  → LVar a d                         -- LVar to listen to
  → (a → IO (Maybe (Par lvl ()))) -- Global callback
  → (d → IO (Maybe (Par lvl ()))) -- Delta callback
  → Par lvl ()
```

As with `getLV`, handlers are specified using both global and delta threshold functions. Rather than returning results, however, these threshold functions return computations to run in a fresh lightweight thread if the threshold has been passed. Each time a callback is launched, the callback count is incremented; when it is finished, the count is decremented, and if zero, all threads blocked on its quiescence are resumed.

The implementation of `addHandler` is very similar to `getLV`, but there is one important difference: handler callbacks must be invoked for *all* events of interest, not just a single threshold. Thus, the `Par` computation returned by the global threshold function should execute its callback on, *e.g.*, all available atoms. Likewise, we do not remove a handler from the bag of listeners when a single delta threshold is passed; handlers listen continuously to an LVar until it is frozen. We might, for example, expose the following `foreach` function for a finite map:

```
foreach mh mapLV cb = addHandler mh lv gThresh dThresh
  where
    dThresh (k,v) = return (Just (cb k v))
    gThresh mp    = traverse mp (λ(k,v) → cb k v) mp
```

Here, idempotence really pays off: without it, we would have to synchronize to ensure that no callbacks are duplicated between the global threshold (which may or may not see concurrent additions to the map) and the delta threshold (which will catch all concurrent additions). We expect such duplications to be rare, since they can only arise when a handler is added concurrently with updates to an LVar.[9]

4.3. Evaluation: k-CFA Case Study

We now evaluate the expressiveness and performance of our Haskell LVish implementation. We expect LVish to particularly shine for: (1) parallelizing complicated algorithms on structured data that pose challenges for other deterministic paradigms, and (2) composing pipeline-parallel stages of computation (each of which may be internally parallelized). In this section, we focus on a case study that fits this mold: *parallelized control-flow analysis*. We discuss the process of porting a sequential implementation of k-CFA to a parallel implementation using LVish. In the companion technical report [**38**], we also give benchmarking results for LVish implementations of

[9]That said, it is possible to avoid all duplication by adding further synchronization, and in ongoing research, we are exploring various locking and timestamp schemes to do just that.

two graph-algorithm microbenchmarks: breadth-first search and maximal independent set.

4.3.1. k-CFA. The k-*CFA* analyses provide a hierarchy of increasingly precise methods to compute the flow of values to expressions in a higher-order language. For this case study, we began with a simple, sequential implementation of k-CFA translated to Haskell from a version by Might [**34**].[10] The algorithm processes expressions written in a continuation-passing-style λ-calculus. It resembles a nondeterministic abstract interpreter in which stores map addresses to *sets* of abstract values, and function application entails a cartesian product between the operator and operand sets. Further, an address models not just a static variable, but includes a fixed k-size window of the calling history to get to that point (the k in k-CFA). Taken together, the current redex, environment, store, and call history make up the abstract state of the program, and the goal is to explore a graph of these abstract states. This graph-exploration phase is followed by a second, summarization phase that combines all the information discovered into one store.

Phase 1: breadth-first exploration. The following function from the original, sequential version of the algorithm expresses the heart of the search process:

```
explore :: Set State → [State] → Set State
explore seen [] = seen
explore seen (todo:todos)
  | todo ∈ seen = explore seen todos
  | otherwise   = explore (insert todo seen)
                          (toList (next todo) ++ todos)
```

[10]Haskell port by Max Bolingbroke: `https://github.com/batterseapower/haskell-kata/blob/master/0CFA.hs`.

This code uses idiomatic Haskell data types like `Data.Set` and lists. However, it presents a dilemma with respect to exposing parallelism. Consider attempting to parallelize `explore` using purely functional parallelism with futures—for instance, using the Strategies library [**31**]. An attempt to compute the next states in parallel would seem to be thwarted by the main thread rapidly forcing each new state to perform the seen-before check, `todo` ∈ `seen`. There is no way for independent threads to "keep going" further into the graph; rather, they check in with `seen` after one step.

We confirmed this prediction by adding a parallelism annotation: `withStrategy (parBuffer 8 rseq) (r`
The GHC runtime reported that 100% of created futures were "duds"—that is, the main thread forced them before any helper thread could assist. Changing `rseq` to `rdeepseq` exposed a small amount of parallelism—238/5000 futures were successfully executed in parallel—yielding no actual speedup.

Phase 2: summarization. The first phase of the algorithm produces a large set of states, with stores that need to be joined together in the summarization phase. When one phase of a computation produces a large data structure that is immediately processed by the next phase, lazy languages can often achieve a form of pipelining "for free". This outcome is most obvious with *lists*, where the head element can be consumed before the tail is computed, offering cache-locality benefits. Unfortunately, when processing a pure `Set` or `Map` in Haskell, such pipelining is not possible, since the data structure is internally represented by a balanced tree whose structure is not known until all elements are present. Thus phase 1 and phase 2 cannot overlap in the purely functional version—but they will in the LVish version, as we will see. In fact, in LVish we will be able to achieve partial deforestation in addition to pipelining. Full deforestation in this application is impossible, because the `Sets` in the implementation serve a memoization purpose: they prevent repeated computations as we traverse the graph of states.

4.3.2. Porting to the LVish Library. Our first step was a *verbatim* port to LVish. We changed the original, purely functional program to allocate a new LVar for each new set or map value in the original code. This was done simply by changing two types, `Set` and `Map`, to their monotonic LVar counterparts, `ISet` and `IMap`. In particular, a store maps a program location (with context) onto a set of abstract values:

```
import Data.LVar.Map as IM
import Data.LVar.Set as IS
type Store s = IMap Addr s (ISet s Value)
```

Next, we replaced allocations of containers, and `map`/`fold` operations over them, with the analogous operations on their LVar counterparts. The `explore` function above was replaced by the simple graph traversal function from Section 3.1! These changes to the program were mechanical, including converting pure to monadic code. Indeed, the key insight in doing the verbatim port to LVish was to consume LVars as if they were pure values, ignoring the fact that an LVar's contents are spread out over space and time and are modified through effects.

In some places the style of the ported code is functional, while in others it is imperative. For example, the `summarize` function uses nested `forEach` invocations to accumulate data into a store map:

```
summarize :: ISet s (State s) → Par d s (Store s)
summarize states = do
  storeFin ← newEmptyMap
  IS.forEach states   $ λ (State _ _ store _) →
    IM.forEach store  $ λ key vals →
```

```
      IS.forEach vals $ λ elmt  →
        IM.modify storeFin key (putInSet elmt)
  return storeFin
```

While this code can be read in terms of traditional parallel nested loops, it in fact creates a network of handlers that convey incremental updates from one LVar to another, in the style of data-flow networks. That means, in particular, that computations in a pipeline can *immediately* begin reading results from containers (*e.g.*, `storeFin`), long before their contents are final.

The LVish version of k-CFA contains 11 occurrences of `forEach`, as well as a few *cartesian-product* operations. The cartesian products serve to apply functions to combinations of all possible values that arguments may take on, greatly increasing the number of handler events in circulation. Moreover, chains of handlers registered with `forEach` result in cascades of events through six or more handlers. The runtime behavior of these would be difficult to reason about. Fortunately, the programmer can largely ignore the temporal behavior of their program, since all LVish effects commute—rather like the way in which a lazy functional programmer typically need not think about the order in which thunks are forced at runtime.

Finally, there is an optimization benefit to using handlers. Normally, to flatten a nested data structure such as `[[[Int]]]` in a functional language, we would need to flatten one layer at a time and allocate a series of temporary structures. The LVish version avoids this; for example, in the code for `summarize` above, three `forEach` invocations are used to traverse a triply-nested structure, and yet the side effect in the innermost handler directly updates the final accumulator, `storeFin`.

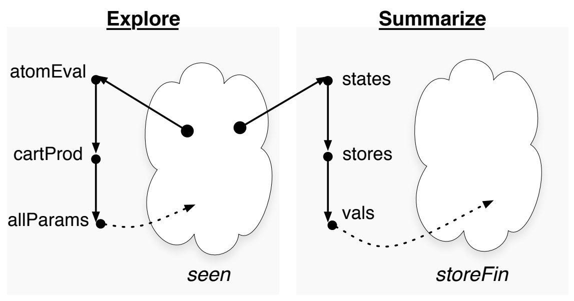

FIGURE 4. Simplified handler network for k-CFA. Exploration and summarization processes are driven by the same LVar. The triply-nested `forEach` calls in `summarize` become a chain of three handlers.

Flipping the switch. The verbatim port uses LVars poorly: copying them repeatedly and discarding them without modification. This effect overwhelms the benefits of partial deforestation and pipelining, and the *verbatim* LVish port has a small performance overhead relative to the original. But not for long! The most clearly unnecessary operation in the verbatim port is in the `next` function. Like the pure code, it creates a fresh store to extend with new bindings as we take each step through the state space graph:

```
store' ← IM.copy store
```

Of course, a "copy" for an LVar is persistent: it is just a handler that forces the copy to receive everything the original does. But in LVish, it is also trivial to *entangle* the parallel branches of the search, allowing them to share information about bindings, simply by *not* creating a copy:

```
let store' = store
```

This one-line change speeds up execution by up to 25× on a single thread, and the asynchronous, `ISet`-driven parallelism enables subsequent parallel speedup as well (up to 202× total improvement over the purely functional version).

Figure 5 shows performance data for the "blur" benchmark drawn from a recent paper on k-CFA [**17**]. (We use $k = 2$ for the benchmarks in this section.) In general, it proved difficult to generate example inputs to k-CFA that took long enough to be candidates for parallel speedup. We were, however, able to "scale up" the blur benchmark by replicating the code N times, feeding one into the continuation argument for the next. Figure 5 also shows the results for one synthetic benchmark that managed to negate the benefits of our sharing approach, which is simply a long chain of 300 "not" functions (using a CPS conversion of the Church encoding for booleans). It has a small state space of large states with many variables (600 states and 1211 variables).

The role of lock-free data structures. As part of our library, we provide lock-free implementations of finite maps and sets based on concurrent skip lists [**26**].[11] We also provide reference implementations that use a nondestructive `Data.Set` inside a mutable container. Our scalable implementation is not yet carefully optimized, and at one and two cores, our lock-free k-CFA is 38% to 43% slower than the reference implementation on the "blur" benchmark. But the effect of scalable data structures is quite visible on a 12-core machine.[12] Without them, "blur" (replicated 8×) stops scaling and begins slowing down slightly after four cores. Even at four cores, variance is high in the reference implementation (min/max 0.96s / 1.71s over 7 runs). With lock-free structures, by contrast, performance steadily improves to a speedup of 8.14× on 12 cores (0.64s at 67% GC productivity). Part of the benefit of LVish is to allow

[11]In fact, this project is the first to incorporate *any* lock-free data structures in Haskell, which required solving some unique problems pertaining to Haskell's laziness and the GHC compiler's assumptions regarding referential transparency. But we lack the space to detail these improvements.
[12]Intel Xeon 5660; full machine details available at `https://portal.futuregrid.org/hardware/delta`.

LK: TODO: Ryan, we should try to make this legible when printed black and white.

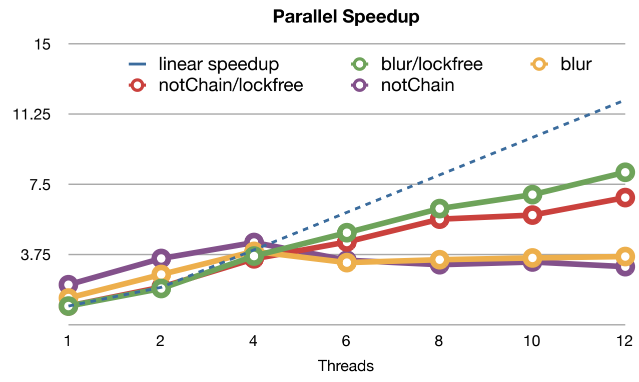

FIGURE 5. Parallel speedup for the "blur" and "notChain" benchmarks. Speedup is normalized to the sequential times for the *lock-free* versions (5.21s and 9.83s, respectively). The normalized speedups are remarkably consistent for the lock-free version between the two benchmarks. But the relationship to the original, purely functional version is quite different: at 12 cores, the lock-free LVish version of "blur" is 202× faster than the original, while "notChain" is only 1.6× faster, not gaining anything from sharing rather than copying stores due to a lack of fan-out in the state graph.

purely functional programs to make use of lock-free structures, in much the same way that the `ST` monad allows access to efficient in-place array computations.

TODO: Add benchmarks and case study from PLDI paper.

CHAPTER 5

Deterministic Threshold Queries of Distributed Data Structures

TODO: Figure out how theorems work in the other chapters.

5.1. Introduction

LK: Got rid of "reusedtext" markings for reused text from the WoDet paper, because we're not worried about those.

Distributed systems typically involve *replication* of data objects across a number of physical locations. Replication is of fundamental importance in such systems: it makes them more robust to data loss and allows for good data locality. But the well-known *CAP theorem* [**23, ?**] of distributed computing imposes a trade-off between *consistency*, in which every replica sees the same data, and *availability*, in which all data is available for both reading and writing by all replicas. *Highly available* distributed systems, such as Amazon's Dynamo key-value store [**16**], relax strong consistency in favor of *eventual consistency* [**53**], in which replicas need not agree at all times. Instead, updates execute at a particular replica and are sent to other replicas later. All updates eventually reach all replicas, albeit possibly in different orders. Informally speaking, eventual consistency says that if updates stop arriving, all replicas will *eventually* come to agree. RRN: Crystal clear so far, but I am now thinking in terms of sending sets of updates rather than sending state snapshots. Is this where you want me to be at? LK: The CRDT people would say that sending sets of updates is the "op-based" style and sending state snapshots is the "state-based" style. They would also say it doesn't really matter, since the two are equivalent. In this paper we're talking about the state-based style, so maybe we should revise this part. ...However, even in the state-based style, the "causal history" is the sets of operations that have taken place on the replicas, *not* the sets of the states they've been in! (Although in the concrete implementation, these sets of operations never get sent.)

Although giving up on strong consistency makes it possible for a distributed system to offer high availability, even an eventually consistent system must have some way of resolving conflicts between replicas that differ. One approach is to try to determine which replica was written most recently, then declare that replica the winner. But, even in the presence of a way to reliably synchronize clocks between replicas and hence reliably determine which replica was written most recently, having the last write win might not make sense from a *semantic* point of view. For instance, if a replicated object represents a set, then, depending on the application, the appropriate way to resolve a conflict between two replicas could be to take the set union of the replicas' contents. Such a conflict resolution policy might be more appropriate than a "last write wins" policy for, say, a object representing the contents of customer shopping carts for an online store [**16**].

5.1.1. Convergent replicated data types and eventual consistency. Implementing application-specific conflict resolution policies in an ad-hoc way for every application is tedious and error-prone.[1] Fortunately, we need not implement them in

[1]Indeed, as the developers of Dynamo have noted [**16**], Amazon's shopping cart presents an anomaly whereby removed items may re-appear in the cart!

an ad-hoc way. Shapiro *et al.*'s *convergent replicated data types* (CvRDTs) [**48, 47**] 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 the lattice's join operation.

In a CvRDT, the contents of a replica can only grow over time—that is, updates must be *inflationary* with respect to the given lattice—and replicas merge with remote replicas by taking the join of the remote state and the local state. CvRDTs offer a simple and theoretically-sound approach to eventual consistency. Still, even with CvRDTs, it is always possible to observe inconsistent *intermediate* states of replicated shared objects, and high availability requires that reads return a value immediately, even if that value is stale.

RRN: The "grow" description is appealing and intuitive. Probably good to leave it there for this paper. But it's good to keep in mind that in some lattices it may require fewer bits to represent higher states of the lattice rather than lower ones. For example, in a series of fork-join diamonds attached vertically, if a higher diamond has a smaller number of branches than a lower one. However, these are also the same kinds of lattices where you can't do the threshold queries that you want. (Because you lose pairwise incompatibility for the sibling elements of the lower diamonds/fork-joins.) LK: Hmmm...we can say something about how we only mean "grow" in a "semantic" sense, not in the sense of actual number of bits it takes to represent a state. RRN: Very low priority issue though. More a note for our benefit.

5.1.2. Strong consistency at the query level. In practice, applications call for both strong consistency and high availability at different times [**?**], and increasingly, they support consistency choices at the granularity of individual queries, not that of the entire system. For example, the Amazon SimpleDB database service gives customers the choice between eventually consistent and strongly consistent read operations on a per-read basis [**?**].

Ordinarily, strong consistency is a global property: all replicas agree on the data. When we make consistency choices at a *per-query* granularity, though, a global strong consistency property need not hold. We define a *strongly consistent query* to be one that, if it returns a result x when executed at a replica i:

- will always return x on subsequent executions at i, and
- will *eventually* return x when executed at *any* replica, and will *block* until it does so.

That is, a strongly consistent query of a distributed data structure, if it returns, will return a result that is a *deterministic* function of all updates to the data structure in the entire distributed execution, regardless of when the query executes or which replica it occurs on.

5.1.3. Our contribution: bringing threshold queries to CvRDTs. As they are today, CvRDTs only support eventually consistent queries. We could get strong consistency by waiting until all replicas agree before allowing a query to return—but in practice, such agreement may never happen. In this paper, we offer an alternative approach to supporting strongly consistent queries that takes advantage of the existing lattice structure of CvRDTs and does not require waiting for all replicas to agree.

To do so we take inspiration from our previous work [**36, 39, 37**] on *LVars*, or lattice-based data structures for shared-memory deterministic parallelism. Like CvRDTs, LVars are data structures whose states are elements of an application-specific lattice, and whose contents can only grow with respect to the given lattice. Unlike CvRDTs, though, LVars make it impossible to observe the *order* of updates to their state.

This is because LVar read operations are *threshold* reads: an attempt to read will block until the data structure's contents reach or surpass a particular "threshold" (which we explain in more detail in Section 5.2), and then return a deterministic result. The combination of inflationary writes and threshold reads allow LVars to serve as the basis for a *deterministic-by-construction* shared-memory parallel programming model: concurrent programs in which all shared data structures are LVars are guaranteed to produce a deterministic outcome on every run, regardless of parallel execution and schedule nondeterminism.LK: "Concurrent programs", but "parallel execution"—seems reasonable to me; will it make sense to other people? RRN: Sounds perfect to me. Readers for whom these are near synonyms will not blink, and readers who make the distinction should be happy too.

Although LVars and CvRDTs were developed independently, both models leverage the mathematical properties of join-semilattices to ensure that a property of the model holds—determinism in the case of LVars; eventual consistency in the case of CvRDTs. Our contribution in this paper is to bring LVar-style *threshold queries* to CvRDTs and show that threshold queries of CvRDTs are strongly consistent queries, according to the criteria given above. After reviewing the fundamentals of threshold queries (Section 5.2) and CvRDTs (Section 5.3), we introduce CvRDTs extended with threshold queries (Section 5.4) and prove that threshold queries in our extended model are strongly consistent queries (Section 5.5). That is, we show that a threshold query that returns an answer when executed on a replica will return the same answer every subsequent time that it is executed on that replica, and that executing that threshold query on a different replica will eventually return the same answer, and will block until it does so. RRN: This is pretty identically repeating what was in the definition in sec 1.2 less than one page earlier, no? LK: yep, just reiterating it – it might be too much. It is therefore impossible to observe different results from the same threshold query, whether at different times on the same replica, or whether on different replicas.

A preliminary version of some of the material in this paper appeared in a non-archival workshop [?]. LK: Things we do in this paper that we didn't do in the WoDet paper: we define what a strongly consistent query actually is; we define threshold queries in a more general way; we take into account the effect of potentially non-terminating threshold queries; we state and prove Theorem 2. Should we list out those things?

RRN: Because wodet allows republication I do not think it is necessary at all to mention this workshop pub. Better to just focus on trumpeting the contributions here. LK: I dunno, I'd feel better mentioning it if we're reusing text from it, even though that's allowed.

RRN: I had mix feelings about them "enforcing" strong consistency... because they don't change the way writes and merges happen. (And if regular reads are still allowed...). What do you think of the above way of casting it? LK: Yeah, that was badly stated. This has been basically overhauled now; we're not talking about proving properties of the whole system, we're talking about proving properties of threshold queries.

LK: I commented out an old footnote about determinism vs. consistency because it's kind of out of date now. Determinism of a system and consistency of a system are different things, but in this paper we're distinguishing between consistency of a *system* and consistency of a *query*, and we're saying that a strongly consistent query is a deterministic query is a threshold query. We aren't talking about determinism of the whole system.

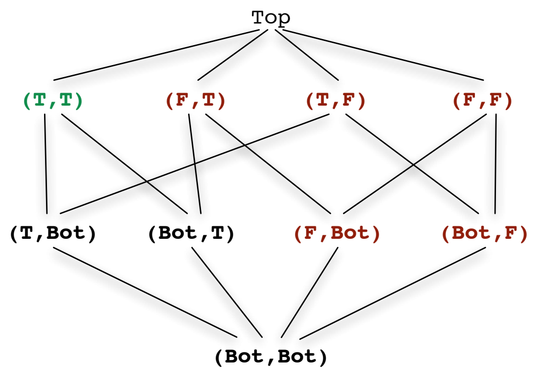

FIGURE 1. Lattice of states that an `AndLV` can take on. The five red states in the lattice correspond to a false result, and the one green to a true one.

5.2. How threshold queries work

A *threshold query* is a way of reading the contents of a lattice-based data structure that allows only limited observations of the state of the data structure. It only returns a value when the data structure's state meets a certain (monotonic) criterion, or "threshold", and the value returned is the same regardless of how far above the threshold the data structure's state goes.

By a *lattice-based data structure*, we mean a data structure whose possible states are elements of a lattice[2], and that can only change over time in a way that is inflationary with respect to the lattice. Both CvRDTs [**48, 47**] and LVars [**36, 39**] are examples of lattice-based data structures. In this section, we illustrate threshold queries with an example before formally describing them in the context of CvRDTs.

5.2.1. An example: parallel "and". Consider a lattice-based data structure that stores the result of a parallel logical "and" operation. We will call this data structure an *`AndLV`*. For this example, we assume two inputs, called "left" and "right", each of

[2]We use "lattice" as a shorthand; formally, the "lattice" of states is a 4-tuple $(D, \sqsubseteq, \bot, \top)$, where D is a set, $\sqsubseteq$ is a partial order on D, $\bot$ is D's least element according to $\sqsubseteq$, $\top$ is D's greatest element, and every two elements in D have a least upper bound. Hence $(D, \sqsubseteq, \bot, \top)$ is really a *bounded join-semilattice* with a designated greatest element of $\top$.

which may be either `T` or `F`. To understand this example, it is sufficient to consider a single replica; in Section 5.4 we will go on to discuss replication and communication between replicas.

RRN: Tweaked this because I didn't want to imply in any way that this example doesn't work with multiple replicas. It's only for simplicity of exposition that we are holding off from talking about those. LK: sounds good.

We can represent the states an `AndLV` can take on as pairs `(x,y)`, where each of `x` and `y` are `T`, `F`, or `Bot`. The `Bot` value, short for "bottom", is the state in which no input has yet been received, and so `(Bot,Bot)` is the least element of the lattice of states that our `AndLV` can take on, shown in Figure 1. An additional state, which we call `Top`, is the greatest element of the lattice; it represents the situation in which an error has occurred—if, for instance, one of the inputs writes `T` and then later changes its mind to `F`.

The result of the parallel "and" computation—if it completes successfully—will be either `True` or `False`, but it might also block indefinitely if not enough writes occur (say, if the left input is `T` and the right input never arrives), or it could end in an error state, discussed below. Each update to the lattice takes the form of a complete state, such as `(T,Bot)` to write the left input. RRN: This section didn't actually describe how write's happen - how is this? LK: oops, fixed now.

The lattice induces a *join*, or least upper bound, operation on pairs of states; for instance, the join of `(T,Bot)` and `(Bot,F)` is `(T,F)`, and the join of `(T,Bot)` and `(F,Bot)` is `Top` since the overlapping `T` and `F` values conflict. Importantly, whenever a write occurs, it updates the `AndLV`'s state to the join of the incoming state and the current state. (With CvRDTs, this join is computed when replicas merge. In this example, though, we are dealing with only one replica, and updating to the join of the new state and the current state is sufficient to ensure that all writes are inflationary.)

LK: TODO: show the Haskell code in an appendix? I don't want to show it in the main part of the paper, not for this audience.

LK: TODO: Maybe we don't want typewriter font here at all, actually...

5.2.2. Threshold queries of an `AndLV`.

Given that we want the result of a threshold query to be a deterministic function of the writes to a data structure—and not of the particular moment in time the query is made—what sorts of observations are safe to make of an `AndLV`? We cannot, for instance, test whether one or both inputs have been written at a particular point in time, because the result will depend on how the test is interleaved with the writes.

Instead, we can describe a *partial function* of the data structure's state that is undefined for all states except those that are at or above a certain *threshold* in the lattice. One way to describe such a function is to define a *threshold set* of sets of "activation" states. In the case of our `AndLV`, one of these sets of activation states is the set of states containing an `F`—that is, the set {`(F,Bot)`, `(Bot,F)`, `(F,T)`, `(T,F)`, `(F,F)`}. The elements of this set are shown in red in Figure 1. The other set of activation states singleton set {`(T,T)`}, shown in green in Figure 1. Therefore the entire threshold set is

$$\{\{\texttt{(F,Bot)},\ \texttt{(Bot,F)},\ \texttt{(F,T)},\ \texttt{(T,F)},\ \texttt{(F,F)}\},\{\texttt{(T,T)}\}\}$$

The semantics of a threshold query is as follows: if a data structure's state reaches (or surpasses) any state or states in a particular set of activation states, the threshold query returns *the entire set* of activation states, regardless of which of those activation states was reached. If no state in any set of activation states has yet been reached, the threshold query will *block*; blocking corresponds to states on which the partial function is undefined.

In the case of our `AndLV`, as soon as either input writes an `F`, our threshold query will unblock and return the first set of activation states, that is, $\{$`(F,Bot)`, `(Bot,F)`, `(F,T)`, `(T,F)`, `(F,F)`$\}$. Hence `AndLV` has the expected "short-circuit" behavior and does not have to wait for a second input if the first input is `F`. If, on the other hand, both inputs write a `T`, the threshold query will unblock and return $\{$`(T,T)`$\}$.

In a real implementation, of course, the value returned from the query could be more meaningful to the client—for instance, we could return `False` instead of returning the set of activation states. However, doing so would only be a convenience, and the translation from $\{$`(F,Bot)`, `(Bot,F)`, `(F,T)`, `(T,F)`, `(F,F)`$\}$ to `False` could just as easily take place on the client side. In either case, the result returned from the threshold query is the same regardless of *which* of the activation states caused it to unblock, and it is impossible for the client to tell whether the actual state of the lattice is, say, `(T,F)` or `(F,F)` or some other state containing `F`.

5.2.3. Pairwise incompatibility. In order for the behavior of a threshold query to be deterministic, it must be unblocked by a *unique* set of activation states in the threshold set. We ensure this by requiring that elements in a set of activation states must be *pairwise incompatible* with elements in every other set of activation states. That is, for all distinct sets of activation states Q and R in a given threshold set: $\forall q \in Q.\ \forall r \in R.\ q \sqcup r = \top$. LK: changed to inline math to save space. In our `AndLV` example, there are two distinct sets of activation states, so if we let $Q = \{$`(T,T)`$\}$ and $R = \{$`(F,Bot)`, `(Bot,F)`, `(F,T)`, `(T,F)`, `(F,F)`$\}$, the least upper bound of `(T,T)` and r must be `Top`, where r is any element of R. We can easily verify that this is the case.

Furthermore, since the lattice of states that an `AndLV` can take on is finite, the join function can be verified to compute a least upper bound.

Why is pairwise incompatibility necessary? Consider the following (illegal) "threshold set" that does not meet the pairwise incompatibility criterion: {{`(F,Bot)`, `(Bot,F)`}, {`(T,Bot)`, `(Bot,T)`

LK: changed to inline math to save space. A threshold query corresponding to this so-called threshold set will unblock and return {`(F,Bot)`, `(Bot,F)`} as soon as a state containing an `F` is reached, and {`(T,Bot)`, `(Bot,T)`} as soon as a state containing a `T` is reached. The trouble with such a threshold query is that there exist states, such as `(F,T)` and `(T,F)`, that could unblock either set of activation states. If the left input writes `F` and the right input writes `T`, and these writes occur in arbitrary order, then the threshold query will return a nondeterministic result, depending on the order in which the two writes arrive. But with the original, pairwise-incompatible threshold set we showed, the threshold query would deterministically return `False`—although if the `T` arrived first, the query would have to block until the `F` arrived, whereas if the `F` arrived first, it could unblock immediately. Hence threshold queries enforce consistency at the expense of availability, but it is still possible to do a "short-circuit" computation that unblocks as soon as an `F` is written.

The threshold set mechanism we describe in this section is part of the LVars programming model discussed in section 5.1; in fact, our `AndLV` example is precisely an LVar [**37**]. But the utility of threshold queries is not limited to LVars. In the following sections, we review the basics of the CvRDT model from the work of Shapiro *et al.*, then show how to add threshold queries to the CvRDT model, and prove that they are strongly consistent queries.

5.3. Background: CvRDTs and eventual consistency

Shapiro *et al.* [**48, 47**] define an *eventually consistent* object as one that meets three conditions. One of these conditions is the property of *convergence*: all correct replicas of an object at which the same updates have been delivered eventually have equivalent state. The other two conditions 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).

RRN: But we kind of break this termination property! The "microops" of individual "t" methods complete immediately, but at the polling layer the "macro-op" can block forever. Do we need to say something about how that is ok because the merges keep happening?

LK: I think that this is actually just fine, because all our proofs are about the inner layer, *not* the polling layer. The argument that we are making is that if the inner layer behaves in the way that we prove it does, then we get determinism by layering a polling layer on top of it – and the polling layer does not have to know anything about threshold sets or lattices or anything like that, so it's very conceptually simple.

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 at which the same updates have been delivered have equivalent state.[3] 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*[4]; or *operation-based* (or "op-based"), also known as *commutative*. CRDTs specified in the state-based style are called *convergent replicated data types*, abbreviated *CvRDTs*, while those specified in the op-based style are called *commutative replicated data types*, abbreviated *CmRDTs*. Of the two styles, we focus on the CvRDT style in this paper because CvRDTs are lattice-based data structures and therefore amenable to threshold queries described in Section 5.2—although, as Shapiro *et al.* show, CmRDTs can emulate CvRDTs and vice versa.

5.3.1. State-based objects. The Shapiro *et al.* model specifies 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, and 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

[3] 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 updates 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.

[4]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.

a single replica, but the merge method m can merge the state from a remote replica with the local replica. The model assumes that each replica sends its state to the other replicas infinitely often, and that eventually every update reaches every replica, whether directly or indirectly.

The assumption that replicas send their state to one another "infinitely often" refers not to the *frequency* of these state transmissions; rather; it says that, regardless of what event (such as an update, via the u method) occurs at a replica, a state transmission is guaranteed to occur after that event. We can therefore conclude that all updates eventually reach all replicas in a state-based object, meeting the "eventual delivery" condition discussed above. However, we still have no guarantee of strong convergence or even convergence. This is where Shapiro *et al.*'s notion of a CvRDT comes in: a state-based object that meets the criteria for a CvRDT is guaranteed to have the strong-convergence property.

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 strongly convergent. Therefore, given the eventual delivery guarantee that all state-based objects have, and given an additional assumption that all method executions terminate, a state-based object that meets the criteria for a CvRDT is SEC [**48**].

5.3.2. Discussion: the need for inflationary updates. LK: I think this is true, but I want someone else to check my reasoning. Although CvRDT updates are required to be inflationary, we note that it is not clear that inflationary updates are necessarily required for convergence. Consider, for example, a scenario in which replicas 1 and 2 both have the state $\{a, b\}$. Replica 1 updates its state to $\{a\}$, a non-inflationary update, and then sends its updated state to replica 2. Replica 2 merges the received state $\{a\}$ with $\{a, b\}$, and its state remains $\{a, b\}$. Then replica 2 sends its state back to replica 1; replica 1 merges $\{a, b\}$ with $\{a\}$, and its state becomes $\{a, b\}$. The non-inflationary update has been lost, and was, perhaps, nonsensical—but the replicas are nevertheless convergent.

However, once we introduce threshold queries of CvRDTs, as we will do in the following section, inflationary updates become *necessary* for the determinism of threshold queries. This is because a non-inflationary update could cause a threshold query that had been unblocked to block again, and so arbitrary interleaving of non-inflationary writes and threshold queries would lead to nondeterministic behavior. Therefore the requirement that updates be inflationary will not only be sensible, but actually crucial.

5.4. Adding threshold queries to CvRDTs

In Shapiro *et al.*'s CvRDT model, 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 threshold queries. In this section, we show how to extend the CvRDT model to accommodate threshold queries.

5.4.1. Objects with threshold queries. Definition 11 extends Shapiro *et al.*'s definition of a state-based object with a threshold query method t:

Definition 11 (state-based object with threshold queries). A *state-based object with threshold queries* (henceforth *object*) is a tuple (S, s^0, q, t, u, m), where S is a set of states, $s^0 \in S$ is the initial state, q is a *query method*, t is a *threshold query method*, u is an *update method*, and m is a *merge method*.

In order to give a semantics to the threshold query method t, we need to formally define the notion of a threshold set described in Section 5.2:

Definition 12 (threshold set). A *threshold set with respect to a lattice* $(S, \leq)$ is a set $\mathcal{S} = \{S_a, S_b, \dots\}$ of one or more sets of *activation states*, where each set of activation states is a subset of S, the set of lattice elements, and where the following *pairwise incompatibility* property holds:

For all $S_a, S_b \in \mathcal{S}$, if $S_a \neq S_b$, then for all activation states $s_a \in S_a$ and for all activation states $s_b \in S_b$, $s_a \sqcup s_b = \top$, where $\sqcup$ is the join operation induced by $\leq$ and $\top$ is the greatest element of $(S, \leq)$.

In our model, we assume a finite set of n processes $p_1, \dots, p_n$, 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 k is ≥ 1 and f is either q, t, u, or m, and a is the arguments to f, if any. Methods execute sequentially at each replica. 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 $q(s) = q(s')$.

5.4.2. 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 13 updates Shapiro *et al.*'s definition of causal history for a state-based object to account for t (a trivial change, since execution of t does not change a replica's causal history):

Definition 13 (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 t, 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.

5.4.3. Threshold CvRDTs and the semantics of blocking. With the previous definitions in place, we can give the definition of a CvRDT that supports threshold queries:

LK: OK, it's a little confusing to use the letter t for the threshold query method, because op-based CRDTs use t for the "prepare-update" method. But, since this paper doesn't have op-based CRDTs, there's no ambiguity...I'm stealing it.

DEFINITION 14 (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 $(S, \leq, s^0, q, t, u, m)$, that has the following properties:

- S forms a join-semilattice ordered by $\leq$.
- S has a greatest element $\top$ according to $\leq$.
- The query method q takes no arguments and returns the local state.
- The threshold query method t takes a threshold set $\mathcal{S}$ as its argument, and has the following semantics: let $t_i^{k+1}(\mathcal{S})$ be the $k+1$th method execution at replica i, where $k \geq 0$. If, for some activation state s_a in some (unique) set of activation states $S_a \in \mathcal{S}$, the condition $s_a \leq s_i^k$ is met, $t_i^{k+1}(\mathcal{S})$ returns the set of activation states S_a. Otherwise, $t_i^{k+1}(\mathcal{S})$ returns the distinguished value block.
- The update method u takes a state as argument and updates the local state to it. RRN: Wait updates the local state to it? Or to the LUB of the input state and it? And if it's the latter, then do we need the next bullet, or is the inflationary property a consequence of this bullet? LK: It actually doesn't do a lub, it just does an inflationary update of any kind. the only lub that happens is with the merge method.
- State is *inflationary* across updates: if u updates a state s to s', then $s \leq s'$.
- The merge method m takes a remote state as its argument, computes the join of the remote state and the local state with respect to $\leq$, and updates the local state to the result.

and the q, t, u, and m methods have no side effects other than those listed above.

We use the block return value to model t's "blocking" behavior as a mathematical function with no intrinsic notion of running duration. When we say that a call to t

"blocks", we mean that it immediately returns block, and when we say that a call to t "unblocks", we mean that it returns a set of activation states S_a.

Modeling blocking as a distinguished value introduces a new complication: we lose determinism, because a call to t at a particular replica may return either block or a set of activation states S_a, depending on the replica's state at the time it is called. However, we can conceal this nondeterminism with an additional layer over the nondeterministic API exposed by t. This additional layer simply *polls* t, calling it repeatedly until it returns a value other than block. Calls to t at a replica that are made by this "polling layer" count as method executions at that replica, and are arbitrarily interleaved with other method executions at the replica, including updates and merges. The polling layer itself need not do any computation other than checking to see whether t returns block or something else; in particular, the polling layer does not need to compare activation states to replica states, since that comparison is done by t itself. RRN: I'm afraid the prev sentence about polling/lvish gives the wrong impression, and a more nuanced one would take more space. So how about we nix it? The problem is that LVish is of course event driven. It doesn't sit around polling gets waiting for puts to happen. It only does anything when the put happens. And yet, the mapping of puts onto exact callbacks is not always perfectly precise. With an IVar it would be, but with an "IMap", the there's a callback registered on a particular key, and yet it might be polled every time a write is made to ANY key. That's just a limitation of the implementation currently – it can't do finer grain lists of guarded callbacks. More generally, some lvars really do *need* polling, yet they only need to poll when a write occurs.

The set of activation states S_a that a call to t returns when it unblocks is unique because of the pairwise incompatibility property described in Definition 12. The intuition behind pairwise incompatibility is that described in Section 5.2.3: without it, different orderings of updates could allow the same threshold query to unblock in different ways, introducing nondeterminism that would be observable beyond the polling layer.

5.4.4. Threshold CvRDTs are strongly eventually consistent. LK: Hmmm. We don't have to do all this blathering about termination anymore, because now threshold queries actually do always return immediately!

We can define eventual consistency and strong eventual consistency exactly as Shapiro *et al.* do in their model. In the following definitions, a *correct replica* is a replica at a correct process, and the symbol $\Diamond$ means "eventually":

DEFINITION 15 (eventual consistency (EC)). An object is *eventually consistent* (EC) if the following three conditions hold:

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

DEFINITION 16 (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: $\forall i, j : c_i = c_j \implies s_i \equiv s_j$.

Since we model blocking threshold queries with block, we need not be concerned with threshold queries not necessarily terminating. Determinism does *not* rule out queries that return block every time they are called (and would therefore cause the polling layer to block forever). However, we guarantee that if a threshold query returns block every time it is called during a complete run of the system, it will do so on *every* run of the system, regardless of scheduling. That is, it is not possible for a query to cause the polling layer to block forever on some runs, but not on others.LK: I changed the wording here from "execution" to "complete run of the system" to lessen confusion with individual method executions.

Finally, we can directly leverage Shapiro *et al.*'s SEC result for CvRDTs to show that a threshold CvRDT is SEC:

THEOREM 1 (strong eventual consistency of threshold CvRDTs). Assuming eventual delivery and termination, 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 [**48**]. Shapiro *et al.*'s proof also assumes that eventual delivery and termination hold for the object, and proves that strong convergence holds — that is, that given causal histories c_i and c_j for respective replicas i and j, that their states s_i and s_j are equivalent. The proof relies on the commutativity of the least upper bound operation. Since, according to our Definition 13, threshold queries do not affect causal history, we can leverage Shapiro *et al.*'s result to say that a threshold CvRDT is also SEC. □ □

RRN: Do we not need to grapple with the blocking forever behavior here with respect to reusing this result off the shelf? LK: right, we don't, because we aren't proving anything about the polling layer, just the inner CvRDT layer.

5.5. Determinism of threshold queries

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 deterministic. For deterministic intermediate results, we must use the threshold query method t. We can show that t is deterministic *without* requiring that the same updates have been delivered at the replicas in question at the time that t runs.

In our previous work on LVars [**36, 39**], we showed that a threshold read of the contents of a shared-memory lattice-based data structure returns a deterministic result regardless of how that query is interleaved with updates to the data structure. Theorem 2 establishes that the same is true for threshold queries of *distributed, replicated* lattice-based data structures—namely, threshold CvRDTs.

THEOREM 2 (determinism of threshold queries). Suppose a given threshold query t on a given threshold CvRDT returns a set of activation states S_a when executed at a replica i. Then, assuming eventual delivery and that no replica's state is ever $\top$ at any point in the execution:

(1) t will always return S_a on subsequent executions at i, and
(2) t will *eventually* return S_a when executed at *any* replica, and will *block* until it does so.

PROOF. The proof relies on transitivity of $\leq$ and eventual delivery of updates. See Appendix 5.8 for the complete proof. □

Although Theorem 2 must assume eventual delivery, it does *not* need to assume strong convergence or even ordinary convergence. It so happens that we have strong convergence as part of strong eventual consistency of threshold CvRDTs (by Theorem 1), but we do not need it to prove Theorem 2. In particular, there is no need for replicas to have the same state in order to return the same result from a particular threshold query. The replicas merely both need to be above an activation state from a unique set of activation states in the query's threshold set. Indeed, the replicas' states may in fact trigger *different* activation states from the same set of activation states.

Theorem 2's requirement that no replica's state is ever $\top$ rules out situations in which replicas disagree in a way that cannot be resolved normally. This would happen if, for instance, updates at two different replicas took their states to activation states from two distinct sets of activation states from the same threshold set. The join of the two replicas' states would be $\top$, which by eventual delivery would become the state of both replicas, but in the meantime, threshold queries of the two replicas would return different results. We rule out this situation by assuming that no replica's state goes to $\top$. (In a replicated version of the parallel "and" example from Section 5.2, this would happen if, for instance, one replica received a write of `T` on its left input while another received a write of `F`, also on its left input—a disagreement for which the only resolution is the error state, `Top`).

5.6. Related work

The extended CvRDT model we present in this paper is based on Shapiro *et al.*'s work on conflict-free replicated data types [**48, 47**], discussed in Section 5.3. Various other authors [**?, 10, 14**] have used lattices as a framework for establishing formal guarantees about eventually consistent systems and distributed programs. Burckhardt *et al.* [**?**] propose a formalism for eventual consistency based on graphs called *revision diagrams*. Burckhardt and Leijen [**10**] show that revision diagrams are semilattices, and Leijen, Burckhardt, and Fahndrich apply the revision diagrams approach to guaranteed-deterministic concurrent functional programming [**29**]. Conway *et al.*'s Bloom^L language for distributed programming leverages the lattice-based semantics of CvRDTs to guarantee confluence [**14**]. The concept of threshold queries comes from our previous work on the LVars model for lattice-based deterministic parallel programming [**36, 39, 37**].

As mentioned in Section 5.1, database services such as Amazon's SimpleDB [**?**] allow for both eventually consistent and strongly consistent reads, chosen at a per-query granularity. Terry *et al.*'s Pileus key-value store [**?**] takes the idea of mixing consistency levels further: instead of requiring the application developer to choose the consistency level of a particular query at development time, the system allows the developer to specify a service-level agreement that can dynamically adapt to changing network conditions, for instance. However, we are not aware of previous work on using lattice-based data structures as a foundation for both eventually consistent and strongly consistent queries.

5.7. Conclusion

In this paper we show how to extend CvRDTs with support for deterministic queries. Borrowing from our previous work on LVars for deterministic parallel programming, we propose *threshold queries*, which rather than returning the exact contents of a CvRDT, only reveal whether the contents have crossed a given "threshold" in the CvRDT's lattice. A threshold query blocks until the threshold is reached and then returns a deterministic result. We prove that a threshold query that returns a particular answer is guaranteed to return the same answer on subsequent queries of that replica, regardless of the replica's state. Moreover, executions of the same query on other replicas are guaranteed to eventually return the same answer, and to block until they do so. The threshold query technique generalizes to any lattice, and hence any CvRDT, allowing CvRDTs to support both eventually consistent and strongly consistent queries without waiting for all replicas to agree.

5.8. Proof of Theorem 2

THEOREM 2 (determinism of threshold queries). Suppose a given threshold query t on a given threshold CvRDT returns a set of activation states S_a when executed at a replica i. Then, assuming eventual delivery and that no replica's state is ever $\top$ at any point in the execution:

(1) t will always return S_a on subsequent executions at i, and
(2) t will *eventually* return S_a when executed at *any* replica, and will *block* until it does so.

PROOF. Consider replica i of a threshold CvRDT $(S, \leq, s^0, q, t, u, m)$. Let $\mathcal{S}$ be a threshold set with respect to $(S, \leq)$. Consider a method execution $t_i^{k+1}(\mathcal{S})$ (*i.e.*, a threshold query that is the $k+1$th method execution on replica i, with threshold set $\mathcal{S}$ as its argument) that returns some set of activation states $S_a \in \mathcal{S}$.

For part 1 of the theorem, we have to show that threshold queries with $\mathcal{S}$ as their argument will always return S_a on subsequent executions at i. That is, we have to show that, for all $k' > (k+1)$, the threshold query $t_i^{k'}(\mathcal{S})$ on i returns S_a.

Since $t_i^{k+1}(\mathcal{S})$ returns S_a, from Definition 14 we have that for some activation state $s_a \in S_a$, the condition $s_a \leq s_i^k$ holds. Consider arbitrary $k' > (k+1)$. Since state is inflationary across updates, we know that the state $s_i^{k'}$ after method execution k'

is at least s_i^k. That is, $s_i^k \leq s_i^{k'}$. By transitivity of $\leq$, then, $s_a \leq s_i^{k'}$. Hence, by Definition 14, $t_i^{k'}(\mathcal{S})$ returns S_a.

For part 2 of the theorem, consider some replica j of $(S, \leq, s^0, q, t, u, m)$, located at process p_j. We are required to show that, for all $x \geq 0$, the threshold query $t_j^{x+1}(\mathcal{S})$ returns S_a eventually, and blocks until it does.[5] That is, we must show that, for all $x \geq 0$, there exists some finite $n \geq 0$ such that

- for all i in the range $0 \leq i \leq n-1$, the threshold query $t_j^{x+1+i}(\mathcal{S})$ returns block, and
- for all $i \geq n$, the threshold query $t_j^{x+1+i}(\mathcal{S})$ returns S_a.

Consider arbitrary $x \geq 0$. Recall that s_j^x is the state of replica j after the xth method execution, and therefore s_j^x is also the state of j when $t_j^{x+1}(\mathcal{S})$ runs. We have three cases to consider:

- $s_i^k \leq s_j^x$. (That is, replica i's state after the kth method execution on i is *at or below* replica j's state after the xth method execution on j.) Choose $n = 0$. We have to show that, for all $i \geq n$, the threshold query $t_j^{x+1+i}(\mathcal{S})$ returns S_a. Since $t_i^{k+1}(\mathcal{S})$ returns S_a, we know that there exists an $s_a \in S_a$ such that $s_a \leq s_i^k$. Since $s_i^k \leq s_j^x$, we have by transitivity of $\leq$ that $s_a \leq s_j^x$. Therefore, by Definition 14, $t_j^{x+1}(\mathcal{S})$ returns S_a. Then, by part 1 of the theorem, we have that subsequent executions $t_j^{x+1+i}(\mathcal{S})$ at replica j will also return S_a, and so the case holds. (Note that this case includes the possibility $s_i^k \equiv s^0$, in which no updates have executed at replica i.)

[5]The occurrences of $k+1$ and $x+1$ in this proof are an artifact of how we index method executions starting from 1, but states starting from 0. The initial state (of every replica) is s^0, and so s_i^k is the state of replica i after method execution k has completed at i.

- $s_i^k > s_j^x$. (That is, replica i's state after the kth method execution on i is *above* replica j's state after the xth method execution on j.)

  We have two subcases:

  - There exists some activation state $s'_a \in S_a$ for which $s'_a \leq s_j^x$. In this case, we choose $n = 0$. We have to show that, for all $i \geq n$, the threshold query $t_j^{x+1+i}(\mathcal{S})$ returns S_a. Since $s'_a \leq s_j^x$, by Definition 14, $t_j^{x+1}(\mathcal{S})$ returns S_a. Then, by part 1 of the theorem, we have that subsequent executions $t_j^{x+1+i}(\mathcal{S})$ at replica j will also return S_a, and so the case holds.
  - There is no activation state $s'_a \in S_a$ for which $s'_a \leq s_j^x$. Since $t_i^{k+1}(\mathcal{S})$ returns S_a, we know that there is some update $u_i^{k'}(a)$ in i's causal history, for some $k' < (k+1)$, that updates i from a state at or below s_j^x to s_i^k.[6] By eventual delivery, $u_i^{k'}(a)$ is eventually delivered at j. Hence some update or updates that will increase j's state from s_j^x to a state at or above some s'_a must reach replica j.[7]

    Let the $x + 1 + r$th method execution on j be the first update on j that updates its state to some $s_j^{x+1+r} \geq s'_a$, for some activation state $s'_a \in S_a$. Choose $n = r + 1$. We have to show that, for all i in the range $0 \leq i \leq r$, the threshold query $t_j^{x+1+i}(\mathcal{S})$ returns block, and that for all $i \geq r + 1$, the threshold query $t_j^{x+1+i}(\mathcal{S})$ returns S_a.

[6]We know that i's state was once at or below s_j^x, because i and j started at the same state s^0 and can both only grow. Hence the least that s_j^x can be is s^0, and we know that i was originally s^0 as well.

[7]We say "some update or updates" because the exact update $u_i^{k'}(a)$ may not be the update that causes the threshold query at j to unblock; a different update or updates could do it. Nevertheless, the existence of $u_i^{k'}(a)$ means that there is at least one update that will suffice to unblock the threshold query.

For the former, since the $x+1+r$th method execution on j is the first one that updates its state to $s_j^{x+1+r} \geq s'_a$, we have by Definition 14 that for all i in the range $0 \leq i \leq r$, the threshold query $t_j^{x+1+i}(\mathcal{S})$ returns block.

For the latter, since $s_j^{x+1+r} \geq s'_a$, by Definition 14 we have that $t_j^{x+1+r+1}(\mathcal{S})$ returns S_a, and by part 1 of the theorem, we have that for $i \geq r+1$, subsequent executions $t_j^{x+1+i}(\mathcal{S})$ at replica j will also return S_a, and so the case holds.

- $s_i^k \not\leq s_j^x$ and $s_j^x \not\leq s_i^k$. (That is, replica i's state after the kth method execution on i is *not comparable* to replica j's state after the xth method execution on j.) Similar to the previous case.

□ □

CHAPTER 6

Related Work

LK: Following stuff is from my proposal.

Work on deterministic parallel programming models is long-standing. As discussed in Section 5.1, 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 [**43**], only allowing single assignments to shared references [**52, 3, 9**], allowing sharing only by a limited form of message passing [**27**], ensuring that concurrent accesses to shared state are disjoint [**7**], resolving conflicting updates after the fact [**29**], or some combination of these approaches. These constraints can be imposed at the language or API level, within a type system, or at runtime.

6.0.1. Deterministic Parallel Java (DPJ). DPJ [**7, 6**] 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.* [**8**] proposed a type and effect system that allows for the incorporation of nondeterministic 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.

6.0.2. FlowPools.

Prokopec *et al.* [**44**] 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.

6.0.3. Concurrent Revisions. The Concurrent Revisions (CR) [**29**] programming model uses isolation types to distinguish regions of the heap shared by multiple mutators. Rather than enforcing exclusive access in the style of DPJ, 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 [**10**] 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).LK: TODO: Write to them and ask them if it's a join-semilattice or a meet-semilattice!

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.

6.0.4. 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 [**2**] 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 [**9**]. More recently, Conway *et al.* [**14**] 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.LK: TODO: make sure this is the case—I think there might be some sort of way to unsafely peek in Bloom, but it's ruled out by the CALM analysis. 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.

LK: Following stuff is from the FHPC paper, and might get axed.

Work on deterministic parallel programming models is long-standing. In addition to the single-assignment and KPN models already discussed, here we consider a few recent contributions to the literature.

Deterministic Parallel Java (DPJ). DPJ [**6**] 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. While a linear type system or region system like that of DPJ could be used to enforce single assignment statically, accommodating λ_{LVar}'s semantics would involve parameterizing the type system by the user-specified lattice.

DPJ also provides a way to unsafely assert that operations commute with one another (using the `commuteswith` form) to enable concurrent mutation. However, DPJ does not provide direct support for modeling message-passing (*e.g.*, KPNs) or asynchronous communication within parallel regions. RRN: DOUBLE CHECK THIS. Finally, a key difference between the λ_{LVar} model and DPJ is that λ_{LVar} retains determinism by restricting *what* can be read or written, rather than by restricting the semantics of reads and writes themselves.

Concurrent Revisions. The Concurrent Revisions (CR) [**29**] 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 policy for resolving conflicts in local copies. The management of shared variables in CR is tightly coupled to a fork-join control structure, and the implementation of these variables is similar to reduction variables in other languages (*e.g.*, Cilk *hyperobjects*). CR charts an important new area in the deterministic-parallelism design space, but one that differs significantly from λ_{LVar}. CR could be used to model similar types of data structures—if versioned variables used least upper bound as their merge function for conflicts—but effects would only become visible at the end of parallel regions, rather than λ_{LVar}'s asynchronous communication within parallel regions.

Bloom and Bloom^L. LK: Augh, I really want to mention CRDTs, but no room!

In the distributed systems literature, *eventually consistent* systems [**54**] leverage the idea of monotonicity to guarantee that, for instance, nodes in a distributed database eventually agree. The Bloom language for distributed database programming [**2**] guarantees eventual consistency for distributed data collections that are updated monotonically. The initial formulation of Bloom had a notion of monotonicity based on *set containment*, analogous to the store ordering for single-assignment languages given in Definition 4. However, recent work by Conway *et al.* [**14**] generalizes Bloom to a more flexible lattice-parameterized system, Bloom^L, in a manner analogous to our generalization from IVars to LVars. Bloom^L comes with a library of built-in lattice types and also allows for users to implement their own lattice types as Ruby classes. Although Conway *et al.* do not give a proof of eventual consistency for Bloom^L, our determinism result for λ_{LVar} suggests that their generalization is indeed safe. Moreover, although the goals of Bloom^L differ from those of LVars, we believe that Bloom^L bodes well for programmers' willingness to use lattice-based data structures, and lattice-parameterized languages based on them, to address real-world programming challenges.

LK: Following stuff is from the POPL paper, and might get axed.

6.1. Related Work

Monotonic data structures: traditional approaches. LVish builds on two long traditions of work on parallel programming models based on monotonically-growing shared data structures:

- In *Kahn process networks* (KPNs) [**27**], as well as in the more restricted *synchronous data flow* systems [**28**], a network of processes communicate with each other through blocking FIFO channels with ever-growing *channel histories*. Each process computes a sequential, monotonic function from the history of its inputs to the history of its outputs, enabling pipeline parallelism. KPNs are the basis for deterministic stream-processing languages such as StreamIt [**24**].

- In parallel *single-assignment languages* [**52**], "full/empty" bits are associated with heap locations so that they may be written to at most once. Single-assignment locations with blocking read semantics—that is, *IVars* [**3**]—have appeared in Concurrent ML as `SyncVars` [**45**]; in the Intel Concurrent Collections system [**9**]; in languages and libraries for high-performance computing, such as Chapel [**12**] and the Qthreads library [**55**]; and have even been implemented in hardware in Cray MTA machines [**5**]. Although most of these uses incorporate IVars into already-nondeterministic programming environments, Haskell's `Par` monad [**32**]—on which our LVish implementation is based—uses IVars in a deterministic-by-construction setting, allowing user-created threads to communicate through IVars without requiring `IO`, so that such communication can occur anywhere inside pure programs.

LVars are general enough to subsume both IVars and KPNs: a lattice of channel histories with a prefix ordering allows LVars to represent FIFO channels that implement a Kahn process network, whereas an LVar with "empty" and "full" states (where $\textit{empty} < \textit{full}$) behaves like an IVar, as we described in Section 3.2. Hence LVars provide a framework for generalizing and unifying these two existing approaches to deterministic parallelism.

Deterministic Parallel Java (DPJ). DPJ [**7, 6**] 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.

LVish 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.* [**8**] proposed a type and effect system that allows for the incorporation of nondeterministic 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, LVish's quasi-determinism arises as a result of schedule nondeterminism.

FlowPools. Prokopec *et al.* [**44**] recently proposed a data structure with an API closely related to ideas in LVish: a FlowPool is a bag 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 LVish, it lacks the ability to detect quiescence, which is crucial for supporting examples like graph traversal, and the `seal` operation is awkward to use when the structure of data is not known in advance. By contrast, our `freeze` operation is more expressive and convenient, but moves the model into the realm of quasi-determinism. Another important difference is the fact that LVish is *data structure-generic*: both our formalism and our library support an unlimited collection of data structures, whereas FlowPools are specialized to bags. Nevertheless, FlowPools represent a "sweet spot" in the deterministic parallel design space: by allowing handlers but not general freezing, they retain determinism while improving on the expressivity of the original LVars model. We claim that, with our addition of handlers, LVish generalizes FlowPools to add support for arbitrary lattice-based data structures.

Concurrent Revisions. The Concurrent Revisions (CR) [**29**] 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 in local copies at join points. Unlike LVish's least-upper-bound writes, CR merge functions are *not* necessarily commutative; the default CR merge function is "joiner wins". Still, semilattices turn up in the metatheory of CR: in particular, Burckhardt and Leijen [**10**] show that, for any two vertices in a CR revision diagram, there exists a *greatest common ancestor* state which 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).

While CR could be used to model similar types of data structures to LVish—if versioned variables used least upper bound as their merge function for conflicts—effects would only become visible at the end of parallel regions, rather than LVish's asynchronous communication within parallel regions. This precludes the use of traditional lock-free data structures as a representation.

Conflict-free replicated data types. In the distributed systems literature, *eventually consistent* systems based on *conflict-free replicated data types* (CRDTs) [**48**] leverage lattice properties to guarantee that replicas in a distributed database eventually agree. Unlike LVars, CRDTs 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. Various data-structure-specific techniques can ensure that non-monotonic updates (such as removal of elements from a set) are not lost.

The Bloom^L language for distributed database programming [**14**] combines 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 LVish, monotonicity is enforced by the LVar API.

Future work will further explore the relationship between LVars and CRDTs: in one direction, we will investigate LVar-based data structures inspired by CRDTs that support non-monotonic operations; in the other direction, we will investigate the feasibility and usefulness of LVar threshold reads in a distributed setting.

TODO: Add stuff from PLDI paper, maybe?

TODO: Add stuff from DISC submission.

Bibliography

[1] Breadth-First Search, Parallel Boost Graph Library. URL `http://www.boost.org/doc/libs/1_51_0/libs/graph_parallel/doc/html/breadth_first_search.html`.

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

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

[4] H. Attiya, R. Guerraoui, D. Hendler, P. Kuznetsov, M. M. Michael, and M. Vechev. Laws of order: expensive synchronization in concurrent algorithms cannot be eliminated. In *POPL*, 2011.

[5] D. A. Bader and K. Madduri. Designing multithreaded algorithms for breadth-first search and st-connectivity on the Cray MTA-2. In *ICPP*, 2006.

[6] R. L. Bocchino, Jr., V. S. Adve, S. V. Adve, and M. Snir. Parallel programming must be deterministic by default. In *HotPar*, 2009.

[7] R. L. Bocchino, Jr., V. S. Adve, D. Dig, S. V. Adve, S. Heumann, R. Komuravelli, J. Overbey, P. Simmons, H. Sung, and M. Vakilian. A type and effect system for deterministic parallel Java. In *OOPSLA*, 2009.

[8] R. L. Bocchino, Jr. et al. Safe nondeterminism in a deterministic-by-default parallel language. In *POPL*, 2011.

[9] Z. Budimlić, M. Burke, V. Cavé, K. Knobe, G. Lowney, R. Newton, J. Palsberg, D. Peixotto, V. Sarkar, F. Schlimbach, and S. Taşirlar. Concurrent Collections. *Sci. Program.*, 18(3-4), Aug. 2010.

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

[11] M. M. T. Chakravarty, R. Leshchinskiy, S. Peyton Jones, G. Keller, and S. Marlow. Data Parallel Haskell: A status report. In *DAMP*, 2007.

[12] B. L. Chamberlain, D. Callahan, and H. P. Zima. Parallel programmability and the Chapel language. *International Journal of High Performance Computing Applications*, 21(3), 2007.

[13] D. Chase and Y. Lev. Dynamic circular work-stealing deque. In *SPAA*, 2005.

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

[15] O. Danvy and A. Filinski. Representing control: a study of the CPS transformation, 1992.

[16] 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.

[17] C. Earl, I. Sergey, M. Might, and D. Van Horn. Introspective pushdown analysis of higher-order programs. In *ICFP*, 2012.

[18] F. Ellen, Y. Lev, V. Luchangco, and M. Moir. SNZI: Scalable NonZero Indicators. In *PODC*, 2007.

[19] M. Felleisen, R. B. Findler, and M. Flatt. *Semantics Engineering with PLT Redex*. The MIT Press, 1st edition, 2009.

[20] M. Fluet, M. Rainey, J. Reppy, and A. Shaw. Implicitly-threaded parallelism in Manticore. In *ICFP*, 2008.

[21] K. Fraser. *Practical lock-freedom*. PhD thesis, 2004.

[22] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the Cilk-5 multithreaded language. In *PLDI*, 1998.

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

[24] M. I. Gordon, W. Thies, M. Karczmarek, J. Lin, A. S. Meli, C. Leger, A. A. Lamb, J. Wong, H. Hoffman, D. Z. Maze, and S. Amarasinghe. A stream compiler for communication-exposed architectures. In *ASPLOS*, 2002.

[25] S. Herhut, R. L. Hudson, T. Shpeisman, and J. Sreeram. River trail: A path to parallelism in javascript. In *OOPSLA*, 2013.

[26] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2008.

[27] G. Kahn. The semantics of a simple language for parallel programming. In J. L. Rosenfeld, editor, *Information processing*. North Holland, Amsterdam, Aug. 1974.

[28] E. Lee and D. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, 1987.

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

[30] C. E. Leiserson, T. B. Schardl, and J. Sukha. Deterministic parallel random-number generation for dynamic-multithreading platforms. In *PPoPP*, 2012.

[31] S. Marlow, P. Maier, H.-W. Loidl, M. K. Aswad, and P. Trinder. Seq no more: better strategies for parallel Haskell. In *Haskell*, 2010.

[32] S. Marlow, R. Newton, and S. Peyton Jones. A monad for deterministic parallelism. In *Haskell*, 2011.

[33] M. M. Michael, M. T. Vechev, and V. A. Saraswat. Idempotent work stealing. In *PPoPP*, 2009.

[34] M. Might. *k*-CFA: Determining types and/or control-flow in languages like Python, Java and Scheme. http://matt.might.net/articles/implementation-of-kcfa-and-0cfa/.

[35] Lindsey Kuper and R. R. Newton. A lattice-theoretical approach to deterministic parallelism with shared state. Technical Report TR702, Indiana University, Oct. 2012. URL `http://www.cs.indiana.edu/cgi-bin/techreports/TRNNN.cgi?trnum=TR702`% fi.

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

[37] Lindsey 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).

[38] Lindsey Kuper, A. Turon, N. R. Krishnaswami, and R. R. Newton. Freeze after writing: Quasi-deterministic parallel programming with LVars. Technical Report TR710, Indiana University, Nov. 2013. URL `http://www.cs.indiana.edu/cgi-bin/techreports/TRNNN.cgi?trnum=TR710`% fi.

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

[40] R. R. Newton and I. L. Newton. PhyBin: binning trees by topology. *PeerJ*, 1:e187, Oct. 2013.

[41] R. S. Nikhil. Id language reference manual, 1991.

[42] P. W. O'Hearn, J. C. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *CSL*, 2001.

[43] S. L. Peyton Jones, R. Leshchinskiy, G. Keller, and M. M. T. Chakravarty. Harnessing the multicores: Nested data parallelism in Haskell. In *FSTTCS*, 2008.

[44] A. Prokopec, H. Miller, T. Schlatter, P. Haller, and M. Odersky. FlowPools: a lock-free deterministic concurrent dataflow abstraction. In *LCPC*, 2012.

[45] J. H. Reppy. *Concurrent Programming in ML*. Cambridge University Press, Cambridge, England, 1999.

[46] D. Sbirlea, K. Knobe, and V. Sarkar. Folding of tagged single assignment values for memory-efficient parallelism. In *Euro-Par*, 2012.

[47] 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.

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

[49] D. Spoonhower, G. E. Blelloch, P. B. Gibbons, and R. Harper. Beyond nested parallelism: tight bounds on work-stealing overheads for parallel futures. In *SPAA*, 2009.

[50] S.-J. Sul and T. L. Williams. A randomized algorithm for comparing sets of phylogenetic trees. In *APBC*, 2007.

[51] D. Terei, D. Mazires, S. Marlow, and S. Peyton Jones. Safe haskell. In *Haskell*, 2012.

[52] L. G. Tesler and H. J. Enea. A language design for concurrent processes. In *AFIPS*, 1968 (Spring).

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

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

[55] K. B. Wheeler, R. C. Murphy, and D. Thain. Qthreads: An API for programming with millions of lightweight threads, 2008.