# Research Statement

Lindsey Kuper

February 17, 2014

The goal of my research is to identify and create tools, techniques, and abstractions that support *compositional reasoning* about programs, proofs, and processes, especially those that do not immediately appear to compose well. To that end, I study programming languages.

For the last few years, I have focused on concurrent and parallel programming models. Parallel tasks updating shared state present a challenge for compositional reasoning. Programs can behave in unexpected ways, or even crash, as a result of unpredictable interactions between parallel tasks. The increasing ubiquity of *irregular* parallel applications—in which the work an algorithm must do, and the potential for parallelizing it, depend on the particular input being processed—exacerbates this challenge: since the work to be done must be dynamically scheduled, unpredictable inter-task interactions at run-time are inevitable. *Deterministic-by-construction* parallel programming models, though, offer the promise of freedom from subtle, hard-to-reproduce bugs caused by schedule nondeterminism at run-time.

In this statement, I discuss my contributions to deterministic-by-construction parallel programming, starting with my work on *lattice-based data structures*, or *LVars* (Section 1), and on combining LVars with other parallel effects in a broadly applicable, guaranteed-deterministic programming model (Section 2). Then, in Section 3, I discuss my ongoing work on the relationship between LVars and *convergent replicated data types* for eventually consistent distributed systems. For brevity, I omit my previous work on *relational programming* [7], *multi-language parametricity* [1], and *testing CPU semantic specifications* [4].

## 1 LVars: lattice-based data structures for deterministic parallelism

Deterministic-by-construction parallel programming models guarantee that programs written using them will have the same observable behavior on every run. To provide that guarantee, though, deterministic-by-construction models must sharply restrict the sharing of state between parallel tasks. Shared state, when not disallowed entirely, is typically restricted to a single type of shared data structure, such as single-assignment locations, known as *IVars* [2, 3], or blocking FIFO queues, as in Kahn process networks [6]. These approaches limit the kinds of deterministic algorithms that can be expressed—efficiently or at all—within the model.

My work in POPL '14 [13] and FHPC '13 [9] introduces *lattice-based data structures*, or *LVars*, which generalize IVars to allow multiple assignments to shared locations. Writes to an LVar update its value to the *least upper bound* of the old and new values, ensuring that its state is monotonically increasing with respect to an application-specific *lattice*. Along with monotonically increasing writes, LVars ensure determinism by allowing only "threshold" reads that block until a lower bound (*i.e.*, an element belonging to a restricted subset of the lattice) is reached. Together, monotonic writes and threshold reads yield a provably deterministic parallel programming model.

Our determinism result is *lattice-generic*: threads can communicate through *any* shared data structure, so long as the states the data structure can take on can be viewed as elements of a lattice and updates are monotonically increasing. This generality allows a programming model based on LVars to subsume existing deterministic parallel programming models. 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 the model with a lattice with one "empty" state and multiple "full" states (where $\forall i.\ \mathit{empty} < \mathit{full}_i$) results in a parallel single-assignment language. Different instantiations of the LVars model result in a wide-ranging family of deterministic parallel languages.

Irregular parallel algorithms, such as graph algorithms, have traditionally presented a challenge for guaranteed-deterministic parallel programming. To address that challenge, LVars support a *freeze* operation, which allows an LVar's contents to be read immediately and exactly without blocking, and *event handlers*, which trigger callbacks 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, which can be used to tell that a fixpoint has been reached. Together, handlers, quiescence, and freezing enable an expressive and useful style of parallel programming; however, freezing imposes the trade-off that, once an LVar has been frozen, any

further writes that would change its value instead raise an exception. We prove that in the presence of freezing, programs are at worst *quasi-deterministic*: on every run, they either produce the same answer or raise an error. The error case can only be caused by a write-after-freeze exception, and error messages can pinpoint the exact racing pair of write and freeze operations so that the programmer can easily correct the bug.

## 2 LVish: deterministic programming with a suite of composable parallel effects

Language-level enforcement of determinism *is* possible, but deterministic-by-construction parallel programming models tend to lack features that would make them applicable to a broad range of real-world problems. Moreover, they lack *extensibility*: it is difficult to add or change language features without breaking the determinism guarantee. Individual language features that are deterministic on their own may not retain determinism when composed.

My work in PLDI '14 ([11], to appear) addresses these extensibility and composability challenges by putting LVars into practice in *LVish* [17], a Haskell library for guaranteed-deterministic parallel programming. The LVish library provides the *`Par` monad*, a mechanism for encapsulating parallel computation, and enables a notion of lightweight, library-level threads to be employed with a custom work-stealing scheduler. LVish provides a variety of lattice-based data structures (*e.g.*, sets, maps, graphs) that support concurrent insertion, but not deletion, during `Par` computations. Users may implement their own lattice-based data structures as well, and LVish provides tools to facilitate the definition of such user-defined LVars.

While applying the LVars model to real problems, we have identified and implemented three capabilities that extend its reach: monotonically increasing updates other than least-upper-bound writes; transitive task cancellation; and parallel mutation of non-overlapping memory locations. The unifying abstraction we use in LVish to add these capabilities without compromising determinism is a form of *monad transformer*, extended to handle the `Par` monad. With these extensions, LVish provides, to our knowledge, the most broadly applicable guaranteed-deterministic parallel programming interface available to date. We have demonstrated the viability of our approach both with traditional parallel benchmarks and with results from a real-world case study: a bioinformatics application [14] that we parallelized using LVish.

An important aspect of programming with LVish is the ability to annotate a `Par` computation with an *effect level*, allowing fine-grained specification of the effects that a given computation is allowed to perform. For instance, since freezing introduces quasi-determinism, a computation annotated with a deterministic effect level is not allowed to perform a 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 an LVar computation, then it is impossible for that freeze to race with a write, ruling out the possibility of a run-time write-after-freeze exception. LVish exposes a way to run `Par` computations that captures this "freeze-last" idiom and has a deterministic effect level.

## 3 Joining forces: LVars and convergent replicated data types

Replication of data across physical locations is of fundamental importance in distributed systems: it makes systems more robust to data loss and allows for good data locality. However, it presents the dilemma of how to resolve conflicts between replicas that differ, particularly in the case of *highly available* distributed systems, such as, for instance, the Dynamo distributed key-value store [5], that give up on strong consistency in favor of *eventual consistency* [18]. Fortunately, *conflict-free replicated data types* (CRDTs) [16] provide a simple mathematical framework for reasoning about and enforcing the eventual consistency of replicated objects in a distributed system. In particular, *state-based* or *convergent* replicated data types, abbreviated as *CvRDTs*, leverage the mathematical properties of lattices to guarantee that all replicas of a distributed object eventually agree.

CvRDTs use lattice properties to ensure eventual consistency across replicas in a way that is closely analogous to how LVars use lattice properties to ensure determinism across parallel executions. Therefore a reasonable next research question is: how can we take inspiration from CvRDTs to improve the LVars model, and vice versa? In my current work ([10], to appear in WoDet '14), I am approaching this question in both directions. In one direction, I consider how LVar-style threshold reads apply to the setting of CvRDTs. Since threshold reads guarantee that the order in which updates occur cannot be observed, they can prevent queries from returning inconsistent intermediate states. Hence lattice-based data structures provide a unified framework for reasoning about and enforcing both eventually consistent *and* strongly consistent queries. In the other direction, I consider how to formally extend the LVars model with CvRDT-style general inflationary updates beyond least-upper-bound writes. Finally, in ongoing work, I am using techniques from the CRDT literature [15] to develop LVars that *emulate* non-monotonic updates, such as sets that support removal of elements and counters that can be decremented as well as incremented, so that programming with lattice-based data structures can be even more broadly applicable.

## References


[1] Amal Ahmed, **Lindsey Kuper**, and Jacob Matthews. Parametric polymorphism through run-time sealing, or, theorems for low, low prices!, February 2011. Northeastern University Programming Languages Seminar talk.

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

[3] Zoran Budimlić, Michael Burke, Vincent Cavé, Kathleen Knobe, Geoff Lowney, Ryan Newton, Jens Palsberg, David Peixotto, Vivek Sarkar, Frank Schlimbach, and Sağnak Taşirlar. Concurrent Collections. *Sci. Program.*, 18(3-4), August 2010.

[4] David Cok, John Phillips, Scott Wisniewski, Suan Hsi Yong, Nathan Lloyd, **Lindsey Kuper**, Denis Gopan, and Alexey Loginov. Safety in numbers. Technical report, GrammaTech, Inc., November 2010.

[5] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. In *SOSP*, 2007.

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

[7] Andrew W. Keep, Michael D. Adams, **Lindsey Kuper**, William E. Byrd, and Daniel P. Friedman. A pattern matcher for miniKanren, or, how to get into trouble with CPS macros. In *Scheme*, 2009.

[8] **Lindsey Kuper** and Ryan R. Newton. A lattice-theoretical approach to deterministic parallelism with shared state. Technical Report TR702, Indiana University, October 2012.

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

[10] **Lindsey Kuper** and Ryan R. Newton. Joining forces: toward a unified account of LVars and convergent replicated data types. In the *5th Workshop on Determinism and Correctness in Parallel Programming (WoDet)*, 2014 (to appear).

[11] **Lindsey Kuper**, Aaron Todd, Sam Tobin-Hochstadt, and Ryan R. Newton. Taming the parallel effect zoo: Extensible deterministic parallelism with LVish. In *PLDI*, 2014 (to appear).

[12] **Lindsey Kuper**, Aaron Turon, Neelakantan R. Krishnaswami, and Ryan R. Newton. Freeze after writing: Quasi-deterministic parallel programming with LVars. Technical Report TR710, Indiana University, November 2013.

[13] **Lindsey Kuper**, Aaron Turon, Neelakantan R. Krishnaswami, and Ryan R. Newton. Freeze after writing: Quasi-deterministic parallel programming with LVars. In *POPL*, 2014.

[14] Ryan R. Newton and Irene L.G. Newton. PhyBin: binning trees by topology. *PeerJ*, 1:e187, October 2013.

[15] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. A comprehensive study of Convergent and Commutative Replicated Data Types. Technical Report RR-7506, INRIA, January 2011.

[16] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In *SSS*, 2011.

[17] Aaron Turon, **Lindsey Kuper**, and Ryan R. Newton. LVish: Parallel scheduler, LVar data structures, and infrastructure to build more. http://hackage.haskell.org/package/lvish, October 2013.

[18] Werner Vogels. Eventually consistent. *Commun. ACM*, 52(1), January 2009.