Freeze After Writing

Quasi-Deterministic Parallel Programming with LVars

Lindsey Kuper
Indiana University
lkuper@cs.indiana.edu

Aaron Turon
MPI-SWS
turon@mpi-sws.org

Neelakantan R. Krishnaswami
University of Birmingham
N.Krishnaswami@cs.bham.ac.uk

Ryan R. Newton
Indiana University
rrnewton@cs.indiana.edu

Abstract

Deterministic-by-construction parallel programming models offer the advantages of parallel speedup while avoiding the nondeterministic, hard-to-reproduce bugs that plague fully concurrent code. A principled approach to deterministic-by-construction parallel programming with shared state is offered by *LVars*: shared memory locations whose semantics are defined in terms of an application-specific lattice. Writes to an LVar take the least upper bound of the old and new values with respect to the lattice, while reads from an LVar can observe only that its contents have crossed a specified threshold in the lattice. Although it guarantees determinism, this interface is quite limited.

We extend LVars in two ways. First, we add the ability to "freeze" and then read the contents of an LVar directly. Second, we add the ability to attach event handlers to an LVar, triggering a callback when the LVar's value changes. Together, handlers and freezing enable an expressive and useful style of parallel programming. We prove that in a language where communication takes place through these extended LVars, programs are at worst *quasi-deterministic*: on every run, they either produce the same answer or raise an error. We demonstrate the viability of our approach by implementing a library for Haskell supporting a variety of LVar-based data structures, together with a case study that illustrates the programming model and yields promising parallel speedup.

Categories and Subject Descriptors D.3.3 [*Language Constructs and Features*]: Concurrent programming structures; D.1.3 [*Concurrent Programming*]: Parallel programming; D.3.1 [*Formal Definitions and Theory*]: Semantics; D.3.2 [*Language Classifications*]: Concurrent, distributed, and parallel languages

Keywords Deterministic parallelism; lattices; quasi-determinism

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 [5, 7, 18, 21, 22, 32], *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 [28], as do languages that force references to be either task-unique or immutable [5]. 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 lines, 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 [18] and StreamIt [16], or shared write-only tables in Intel Concurrent Collections [7].

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 [19] makes an initial foray into programming with monotonic data structures. In this model (which we review in Section 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

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

POPL '14, January 22–24, 2014, San Diego, CA, USA.

Copyright is held by the owner/author(s). Publication rights licensed to ACM.

ACM 978-1-4503-2544-8/14/01...$15.00.
http://dx.doi.org/10.1145/2535838.2535842

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.

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)
  freezeSetAfter seen handle
```

This code, written using our Haskell implementation (described in Section 6),[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.

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). In addition to our high-level design, we present a core calculus for LVish (Section 4), formalizing its semantics, and include a runnable version, implemented in PLT Redex (Section 4.7), for interactive experimentation.
- We give a proof of quasi-determinism for the LVish calculus (Section 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 6). 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 7, we evaluate our library with a case study: parallelizing control flow analysis. The case study begins with an existing implementation of k-CFA [26] 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

[1] Our original work on LVars [19] 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.

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

[3] The same is true for quiescence detection; see Section 3.2.

yields promising parallel speedup, and also turns out to have benefits even in the sequential case.

2. Background: the LVars Model

IVars [1, 7, 24, 27] are a well-known mechanism for deterministic parallel programming. An IVar is a *single-assignment* variable [32] 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* [19] 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:

$$\begin{array}{ll} \texttt{let par } _ = \texttt{put } lv\ 3 & \\ \qquad\qquad _ = \texttt{put } lv\ 2 & \text{(Example 1)} \\ \quad \texttt{in get } lv & \end{array}$$

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

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

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 1 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 1 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 1 to allow the get operation to be interleaved with the two puts:

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

Since the two puts and the get can be scheduled in any order, Example 2 is nondeterministic: 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

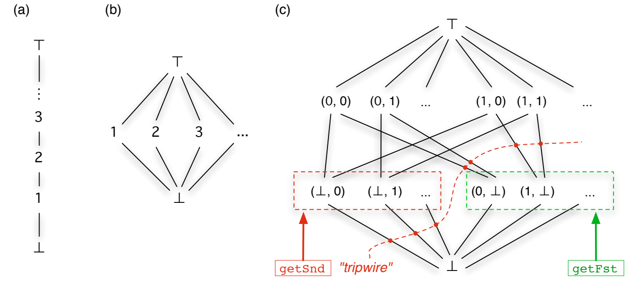

Figure 1. Example LVar lattices: (a) positive integers ordered by $\leq$; (b) IVar containing a positive integer; (c) pair of natural-number-valued IVars, annotated with example threshold sets that would correspond to a blocking read of the first or second element of the pair. Any state transition crossing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

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 2 behave deterministically by passing a threshold set argument to get. 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.

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:

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

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

This allows us to write programs like the following:

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

In the call getSnd lv, the threshold set is $\{(\bot, 0), (\bot, 1), \ldots\}$, 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 4.5.)

In the case of Example 3, `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 6) 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. LVish, Informally

As we explained in Section 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 4 then formalizes them, while Section 6 shows how to implement them.

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

$$\texttt{addHandler}\ lv\ \{1, 3, 5, \dots\}\ (\lambda x.\ \texttt{put}\ lv\ x+1) \quad \text{(Example 4)}$$

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 4 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:

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

(Example 5)

Can Example 5 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 `put`s. 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 5, 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 7.

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 6.2, we will see that this kind of handler set can be specified in a lattice-generic way, and in Section 6 we will see that it corresponds closely to our implementation strategy.

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 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:

$$\begin{aligned}&\texttt{let}\ h = \texttt{newPool}\\ &\quad \texttt{in addInPool}\ h\ lv\ Q\ f;\\ &\qquad \texttt{quiesce}\ h\end{aligned}$$

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 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 1, `puts` 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:

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

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 1.

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 1, are given in the companion technical report [20].

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.

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 1 and Lemma 1 formalize this notion.

Definition 1 (Lattice freezing)**.** Suppose $(D, \sqsubseteq, \bot, \top)$ is a lattice. We define an operation $\text{Freeze}(D, \sqsubseteq, \bot, \top) \stackrel{\triangle}{=} (D_p, \sqsubseteq_p, \bot_p, \top_p)$ as follows:

1. D_p is a set defined as follows:
$$D_p \stackrel{\triangle}{=} \{(d, frz) \mid d \in (D - \{\top\}) \wedge frz \in \{\mathsf{true}, \mathsf{false}\}\} \cup \{(\top, \mathsf{false})\}$$
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. $\bot_p \stackrel{\triangle}{=} (\bot, \mathsf{false})$.
4. $\top_p \stackrel{\triangle}{=} (\top, \mathsf{false})$.

Lemma 1 (Lattice structure)**.** *If* $(D, \sqsubseteq, \bot, \top)$ *is a lattice then* $\text{Freeze}(D, \sqsubseteq, \bot, \top)$ *is as well.*

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

We use the notation $S[l \mapsto (d, frz)]$ to denote extending S with a binding from l to (d, frz). If $l \in dom(S)$, then $S[l \mapsto (d, frz)]$ 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 $[l_1 \mapsto (d_1, frz_1), l_2 \mapsto (d_2, frz_2), \ldots]$.

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

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

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

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

Figure 2. Syntax for LVish.

Definition 3. 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 [20]). If, for example,

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

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 4.5, an attempted `put` that would take the value of l to $\top$ will raise an error.

4.4 The LVish Calculus

The syntax and operational semantics of the LVish calculus appear in Figures 2 and 3, 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:

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

[4] This is in contrast to the original LVars formalism given in [19], which models parallelism with explicitly simultaneous reductions.

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.

4.5 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 $(\bot, \textsf{false})$, 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 $(d_2, \textsf{false})$, 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 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.

4.6 The `freeze − after − with` Primitive

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. 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`).

[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 6.

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

$incomp(P) \triangleq \forall p_1, p_2 \in P.\ (p_1 \neq p_2 \implies p_1 \sqcup_p p_2 = \top_p)$

$\boxed{\sigma \hookrightarrow \sigma'}$

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

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

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

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

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 LVish.

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 `puts` that would have enabled additional callbacks will instead fault, raising **error** by way of the E-PUT-ERR rule.

Therefore, freezing is a way of "betting" that once a collection of callbacks have completed, no further `puts` that change the LVar's value will occur. For a given run of a program, either all `puts` to an LVar arrive before it has been frozen, in which case the value returned by `freeze − after − with` is the lub of those values, or some `put` arrives after the LVar has been frozen, in which case the program will 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 `puts` either prior to `freeze − after − with`, or by the callback function within it (as will be the case for fixpoint computations). In practice, freezing is usually the 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.

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 [14]. 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 3, 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 `freeze − after − with`, in particular, to determine when the E-FREEZE-FINAL rule can fire);
- 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 ques-

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

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

tion. 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.

5. Quasi-Determinism for LVish

Our proof of quasi-determinism for LVish formalizes the claim we make in Section 1: that, for a given program, although some executions may raise exceptions, all executions that produce a final result will produce the same final result.

In this section, 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 [20].

5.1 Quasi-Determinism and Quasi-Confluence

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

Theorem 1 (Quasi-Determinism)**.** *If $\sigma \hookrightarrow^* \sigma'$ 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 1 follows from a series of *quasi-confluence* lemmas. The most important of these, Strong Local Quasi-Confluence (Lemma 2), 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 2's result to multiple steps by induction on the number of steps, eventually building up to Theorem 1.

Lemma 2 (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**.*

5.2 Independence

In order to show Lemma 2, we need a "frame property" for LVish that captures the idea that independent effects commute with each other. Lemma 3, 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. 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 3 (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 3 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 4. 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 5. A store S'' is *non-conflicting* with the transition $\langle S; e\rangle \hookrightarrow \langle S'; e'\rangle$ iff $(dom(S') - dom(S)) \cap dom(S'') = \emptyset$.

6. 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`

Our library adopts the basic approach of the `Par` monad [24], 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 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:[8]

```
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
```

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.[9]

[8] 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`.

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

6.1 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, 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 [17]. 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 [2]. And in any case, monotonic data structures are usually much simpler to represent and implement than general ones.

6.2 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 `puts` and `gets`/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 `puts` or `freezes` 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 [25]: 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 [10]). But idempotence also helps us deal with races between `put` and `get`/`addHandler`, as we explain below.

6.3 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.[10] 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 `gets` and handlers; once the LVar is frozen, there can be no further events to listen for.[11] 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). It is also given access to the CPU-local scheduler queue, which it can use to spawn threads.

6.4 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 4 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

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

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

```
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
  delta   ← doPut state       -- possibly modify LVar
  curStat ← readIORef status -- read while q is marked
  Sched.clearMark q           -- retract our intent
  whenJust delta $ λd → do
    case curStat of
      Frozen → error "Attempt to change a frozen LVar"
      Active listeners → B.foreach listeners $
        λ(Listener onUpd _) tok → onUpd d tok q
  k () q

freezeLV :: LVar a d → Par QuasiDet ()
freezeLV (LVar {status}) = mkPar $ λk q → do
  Sched.awaitClear q
  oldStat ← atomicModifyIORef status $ λs→(Frozen, s)
  case oldStat of
    Frozen → return ()
    Active listeners → B.foreach listeners $
      λ(Listener _ onFrz) tok → onFrz tok q
  k () q
```

Figure 4. Implementation of key lattice-generic functions.

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
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.[12]

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, 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 `put`s that *caused* the freeze to take place check the frozen status *before* the freeze takes place; additional `put`s 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

[12] 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 [15].

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.[13] 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.

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.[14]

7. 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 [20], we also give benchmarking results for LVish implementations of two graph-algorithm microbenchmarks: breadth-first search and maximal independent set.

7.1 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 [26].[15] 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)
```

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 [23]. An attempt to compute the next states in parallel would seem to be thwarted by the 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) (next todo)`. 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

[13] One can use a high-performance *scalable non-zero indicator* [13] to implement `Counter`, but we have not yet done so.

[14] 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.

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

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.

7.2 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 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 dataflow 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

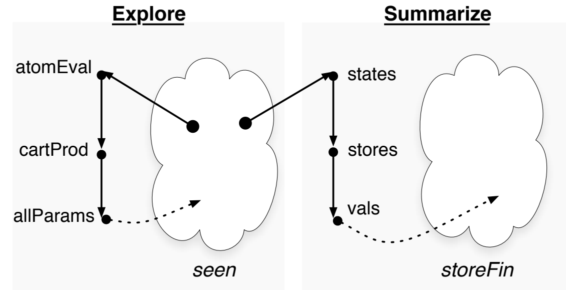

Figure 5. Simplified handler network for k-CFA. Exploration and summarization processes are driven by the same LVar. The triply-nested `forEach` calls in `summarize` become a chain of three handlers.

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`.

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 $25\times$ on a single thread, and the asynchronous, `ISet`-driven parallelism enables subsequent parallel speedup as well (up to $202\times$ total improvement over the purely functional version).

Figure 6 shows performance data for the "blur" benchmark drawn from a recent paper on k-CFA [12]. (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 6 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 [17].[16] We also provide reference implementations that use a nondestructive `Data.Set` inside a mutable container.

[16] In fact, this project is the first to incorporate *any* lock-free data structures in Haskell, which required solving some unique problems pertaining

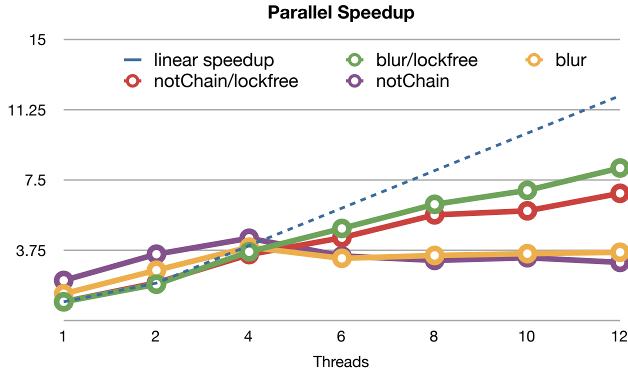

Figure 6. Parallel speedup for the "blur" and "notChain" benchmarks. Speedup is normalized to the sequential times for the *lock-free* versions (5.21s and 9.83s, respectively). The normalized speedups are remarkably consistent for the lock-free version between the two benchmarks. But the relationship to the original, purely functional version is quite different: at 12 cores, the lock-free LVish version of "blur" is $202\times$ faster than the original, while "notChain" is only $1.6\times$ faster, not gaining anything from sharing rather than copying stores due to a lack of fan-out in the state graph.

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.[17] Without them, "blur" (replicated $8\times$) 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 $8.14\times$ on 12 cores (0.64s at 67% GC productivity). Part of the benefit of LVish is to allow 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.

8. 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) [18], as well as in the more restricted *synchronous data flow* systems [21], 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 [16].
- In parallel *single-assignment languages* [32], "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* [1]—have appeared in Concurrent ML as `SyncVars` [30]; in the Intel Concurrent Collections system [7]; in languages and libraries for high-performance computing, such as Chapel [9] and the Qthreads library [33]; and have even been implemented in hardware in Cray MTA machines [3]. Although most of these uses incorporate IVars into already-nondeterministic programming environments, Haskell's `Par` monad [24]—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 $empty < full$) behaves like an IVar, as we described in Section 2. Hence LVars provide a framework for generalizing and unifying these two existing approaches to deterministic parallelism.

Deterministic Parallel Java (DPJ) DPJ [4, 5] 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.* [6] 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.* [29] 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.

to Haskell's laziness and the GHC compiler's assumptions regarding referential transparency. But we lack the space to detail these improvements.

[17] Intel Xeon 5660; full machine details available at `https://portal.futuregrid.org/hardware/delta`.

Concurrent Revisions The Concurrent Revisions (CR) [22] 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 [8] 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) [31] 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 [11] 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.

Acknowledgments

Lindsey Kuper and Ryan Newton's work on this paper was funded by NSF grant CCF-1218375.

References

[1] Arvind, R. S. Nikhil, and K. K. Pingali. I-structures: data structures for parallel computing. *ACM Trans. Program. Lang. Syst.*, 11, October 1989.

[2] 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.

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

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

[5] 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.

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

[7] 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, August 2010.

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

[9] 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.

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

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

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

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

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

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

[16] 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.

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

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

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

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

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

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

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

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

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

[26] 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/.

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

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

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

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

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

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

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