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
MPI-SWS
neelk@mpi-sws.org

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

Abstract

Deterministic-by-construction parallel programming models offer programmers the promise of freedom from subtle, hard-to-reproduce nondeterministic bugs in parallel 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 a user-specified 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 callback functions to an LVar, allowing events to be triggered by writes to it. Together, callbacks and freezing enable an expressive and useful style of parallel programming. We prove that in a language where communication takes place through freezable 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 two case studies that illustrate the programming model and yield promising parallel speedup.

1. Introduction

Nondeterminism is essential for achieving flexible parallelism: it allows tasks to be scheduled onto cores dynamically, in response to the vagaries of an execution. But if 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 programs in the first place.

While much work has focused on identifying methods of deterministic parallel programming [5, 6, 14, 17, 18, 26], *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 [13], 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 centered around a single data structure: for instance, FIFO queues in Kahn process networks [14] and StreamIt [11], or shared write-only tables in Intel Concurrent Collections [6].

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 languages. 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.

The *LVars* programming model recently proposed by Kuper and Newton makes an initial foray into programming with monotonic data structures [16]. In their model (which we review in Section 2), all shared data structures (called LVars) are monotonic, and each one is associated with a *lattice* from which its states are drawn. 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 LVar 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 goes over 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 LVar model guarantees determinism, supports an unlimited variety of data structures (anything viewable as a lattice), and provides a familiar API, so it already achieves several of our goals. Unfortunately, it is not as general-purpose as one might hope.

Many algorithms are presented explicitly as fixpoints of monotonic functions. For example, an unordered graph traversal can be understood in terms of a monotonically growing set of "seen nodes"; neighbors of seen nodes are fed back into to the set until it reaches a fixed point. Similar algorithms are common enough in static analysis to warrant a generic "Monotone Framework" for expressing them [15]. These kinds of algorithms would seem to be a perfect match for the LVar model, but they are not expressible using the monotonic writes and reads described above.

The problem is that these algorithms rely on *negative* information. In a graph traversal, for example, neighboring nodes should only be explored if the current node has not been seen yet; 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 LVar model, asking whether a node is in a set means waiting until the node is in the set.

Monotonic data structures that can say "no" In this paper, we propose two additions to the LVar 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 "pull" model of programming in which threads ask specific questions of an LVar and block until they receive an answer. Handlers, by contrast, support a "push" model of programming in which actions are driven by asynchronous events, in the spirit of data flow programming. Crucially, it is possible to ask a negative question about a handler—"are any callbacks ready to run?"—which makes it possible to tell that a fixpoint has been reached (*i.e.*, there are no further changes to respond to).

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, asking both positive and negative questions 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 (see below),[2] discovers the set of nodes reachable from a given starting node, in parallel, and is guaranteed to be deterministic. It works by creating a fresh set LVar (corresponding to a lattice 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. 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 freeze 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 freeze/write pair that are racing, greatly easing debugging. Of course, some nondeterministic exceptions might never show up in testing, but quasi-determinism is also helpful for moving into production: in many cases, a programmer can choose the defensive technique of catching the exception and retrying the computation (perhaps from a snapshot), under the assumption that the exception is unlikely to occur again.

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, waiting until all other threads have completed. 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 $p' \sqcup p_F$. Independence 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 valid, monotonic LVar interface. Our library exposes a monad that is *indexed* by a determinism level: fully- or quasi-deterministic. Thus, the *static type* of an LVish computation reflects its guarantee, and in particular the freeze-last idiom allows freeze to be used safely with a fully-deterministic index.
- In Section 7, we evaluate our library in the context of two application domains: graph algorithms and control flow analysis. We use graph benchmarks from the Problem Based Benchmark Suite [25] and an algorithm for k-CFA proposed by Might [21]. These two domains illustrate the advantage of LVars over more limited forms of deterministic parallelism.

2. Background: the LVars Model

IVars [2, 6, 19, 22] are a well-known mechanism for deterministic parallel programming. An IVar is a *single-assignment* variable [26] with a blocking read semantics: attempts to read an empty IVar will block until it has been filled with a value. Kuper and Newton [16] proposed *LVars* 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 a user-specified lattice of states.

Consider a program in which two parallel computations write to a natural-number-valued LVar lv, with one thread writing the value 2 and the other writing 3:

$$\texttt{let par } \_ = \texttt{put } lv\ 3;\ \_ = \texttt{put } lv\ 2 \texttt{ in get } lv \qquad \text{(Example 1)}$$

[1] Kuper and Newton briefly sketched 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 effectively program with it in conjunction with our other proposal, handlers.

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

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

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

launches parallel subcomputations $e_1, e_2, \ldots$ that run in arbitrary (interleaved) order, but that all complete before *body* runs. The `put` operation is defined in terms of the user-specified lattice of LVar states: it updates the LVar to the *least upper bound* of the current state and the new state. Supposing that the user-specified lattice is the usual $\leq$ ordering on natural numbers (in which the least upper bound of two natural numbers n_1 and n_2 is $\max(n_1, n_2)$), lv's state will always be $\max(3, 2) = 3$ by the time `get` lv runs. Therefore Example 1 will deterministically evaluate to 3, regardless of the order in which the two `put` operations occurred.

Threshold reads However, merely ensuring that writes to an LVar are monotonically increasing is not enough to ensure that programs are deterministic. For example, suppose we change Example 1 to allow `get` lv to be interleaved with the two `put`s:

$$\texttt{let par}\ \_ = \texttt{put}\ lv\ 3;\ \_ = \texttt{put}\ lv\ 2;\ x = \texttt{get}\ lv\ \texttt{in}\ x \qquad \text{(Example 2)}$$

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

We can make Example 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 safely share them between threads without introducing unintentional 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 does not seem particularly useful: 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. 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:

$$\texttt{let par}\ \_ = \texttt{put}\ lv\ (\bot, 4);\ \_ = \texttt{put}\ lv\ (3, \bot);\ x = \texttt{getSnd}\ lv\ \texttt{in}\ x \qquad \text{(Example 3)}$$

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 $\{(\bot, 0), (\bot, 1), \ldots\}$ can ever be reached. (We formalize this incompatibility requirement in Section 4.5.) Therefore, for any given program, `get` will behave deterministically regardless of the size of the threshold set, so long as the elements of the threshold set are incompatible. 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 use of threshold sets in the LVars model should be understood as a mathematical modeling technique, *not* an implementation approach or practical API. Our library (discussed in Section 6) exposes an unsafe `getLV` operation to the authors of data structure libraries, who can then make operations like `getSnd` available as a safe interface for application writers, baking in the particular thresholds that make sense for a given data structure without ever explicitly constructing them. To put it another way, read operations on a data structure exposed as an LVar must *semantically* behave as threshold reads, but the threshold sets may be fixed in advance and invisible to the client.

3. LVish, Informally

As we explained in Section 1, while LVars offer a deterministic programming model that covers a wide range 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 is 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. So, for example, if lv is an LVar drawn from the lattice of natural numbers, the expression

$$\texttt{addHandler}\ lv\ \{0, 2, 4, \ldots\}\ (\lambda x.\ \texttt{put}\ lv\ x + 1) \qquad \text{(Example 4)}$$

registers a handler for lv that executes a callback for each even number that lv is above; the callback puts the next largest odd number back into the LVar.

In general, the second argument to `addHandler` is an arbitrary subset of the LVar's lattice, Q, saying 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. Handlers in LVish 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. So, if Example 4 is sequenced after `put` lv 4, its callback will execute once for each of the values 0, 2, and 4. To see why this semantics is necessary, consider the following, more subtle example:

$$\begin{array}{l}\texttt{let par}\ \_ = \texttt{put}\ lv\ 0;\ \_ = \texttt{put}\ lv\ 1 \\ \qquad\qquad \_ = \texttt{addHandler}\ lv\ \{0, 1\}\ (\lambda x.\ \texttt{if}\ x = 0\ \texttt{then put}\ lv\ 2) \\ \quad \texttt{in get}\ lv\ \{2\}\end{array} \qquad \text{(Example 5)}$$

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 has occurred, then the example would be

nondeterministic: it would block, or not, depending on how the handler registration was interleaved with the `puts`. By instead executing a handler's callback once for *each and every* element in its event set below 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 consider the lattice of sets of natural numbers, 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 general pattern of handler pool usage is as follows:

$$\texttt{let}\ h\ =\ \texttt{newPool in addInPool}\ h\ lv\ Q\ f;\ \texttt{quiesce}\ h$$

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

Of course, whether or not a handler is quiescent is a non-monotonic property: we can move in and out of quiescence as more `puts` 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 `puts` 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`. But 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 lattice instead raise an exception, and it is the potential for races between such `puts` and `freeze` that makes LVish quasi-deterministic, rather than fully deterministic.

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

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

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

- We parameterize the definition of the LVish calculus by a *single* user-specified lattice, representing the set of states that LVars in the calculus can take on, which means that 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 implementation supports multiple lattices natively.
- Rather than modeling the full ensemble of event handlers, handler pools, quiesce, and freeze 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.

4.1 Lattices

The user-specified lattice is given as 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.

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, an LVar's *state* is a pair (d, frz), where d is an element of the user-provided 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 user-provided ordering $\sqsubseteq$ on elements of D. Every element of D is "freezable" except $\top$. Informally:

- Two unfrozen states are ordered according to the user-specified $\sqsubseteq$; 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 $\top$: $(d, \mathsf{true}) \sqsubseteq_p (d', \mathsf{false})$ exactly when $d' = \top$.

The addition of status bits to the user-specified lattice results in a new lattice $(D_p, \sqsubseteq_p, \bot_p, \top_p)$. Definitions 1 and 2 and Lemma 1 formalize this notion. (The proof of Lemma 1 is included in the non-anonymous supplemental material submitted with this paper.)

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

1. D_p is a set defined as follows:

$$D_p \triangleq \{(d, frz) \mid d \in (D - \{\top\}) \wedge frz \in \{\mathsf{true}, \mathsf{false}\}\} \cup \{(\top, \mathsf{false})\}$$

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

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

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

Definition 2. We define a binary operator $\sqcup_p \in D_p \times D_p \to D_p$ as follows:

$$\begin{array}{lcl} (d_1, \mathsf{false}) \sqcup_p (d_2, \mathsf{false}) & = & (d_1 \sqcup d_2, \mathsf{false}) \\ (d_1, \mathsf{true}) \sqcup_p (d_2, \mathsf{true}) & = & \begin{cases} (d_1, \mathsf{true}) & \text{if } d_1 = d_2 \\ (\top, \mathsf{false}) & \text{otherwise} \end{cases} \\ (d_1, \mathsf{false}) \sqcup_p (d_2, \mathsf{true}) & = & \begin{cases} (d_2, \mathsf{true}) & \text{if } d_1 \sqsubseteq d_2 \\ (\top, \mathsf{false}) & \text{otherwise} \end{cases} \\ (d_1, \mathsf{true}) \sqcup_p (d_2, \mathsf{false}) & = & \begin{cases} (d_1, \mathsf{true}) & \text{if } d_2 \sqsubseteq d_1 \\ (\top, \mathsf{false}) & \text{otherwise} \end{cases} \end{array}$$

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. Although each LVar in a program has its own state, the states of all the LVars are drawn from the same lattice $(D_p, \sqsubseteq_p, \bot_p, \top_p)$. We can do this with no loss of generality because lattices corresponding to different types of LVars could always be unioned into a single lattice (with shared $\bot_p$ and $\top_p$ elements). Alternatively, in a typed formulation of LVish, the type of an LVar might determine the lattice of its states.

Definition 3. A *store* is either a finite partial mapping $S : Loc \xrightarrow{\text{fin}} (D_p - \{\top_p\})$, 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 $\sqsubseteq_p$ and $\sqcup_p$ operations defined on elements of D_p to the level of stores:

Definition 4. A store S is *less than or equal to* a store S' (written $S \sqsubseteq_S S'$) iff:

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

Definition 5. The *least upper bound (lub)* of two stores S_1 and S_2 (written $S_1 \sqcup_S S_2$) is defined as follows:

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

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

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

Figure 1. Syntax for LVish.

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

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

By Definition 5, if, for example,

$$(d_1, frz_1) \sqcup_p (d_2, frz_2) = \top_p,$$

then

$$[l \mapsto (d_1, frz_1)] \sqcup_S [l \mapsto (d_2, frz_2)] = \top_S.$$

Notice that a store containing a binding $l \mapsto (\top, frz)$ 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 LVish appear in Figures 1 and 2, respectively. As we have noted, both the syntax and semantics are parameterized by the lattice $(D, \sqsubseteq, \bot, \top)$. The reduction relation $\longhookrightarrow$ 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.[3]

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 a syntactic sugar `let par`

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

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

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

$\boxed{\sigma \longrightarrow \sigma'}$

$$\text{E-EVAL-CTXT}\ \frac{\langle S;\, e\rangle \longrightarrow \langle S';\, e'\rangle}{\langle S;\, E[e]\rangle \longrightarrow \langle S';\, E[e']\rangle} \qquad \text{E-BETA}\ \frac{}{\langle S;\, (\lambda x.\, e)\ v\rangle \longrightarrow \langle S;\, e[x := v]\rangle} \qquad \text{E-NEW}\ \frac{}{\langle S;\, \texttt{new}\rangle \longrightarrow \langle S[l \mapsto (\bot, \mathsf{false})];\, l\rangle}\ (l \notin dom(S))$$

$$\text{E-PUT}\ \frac{S(l) = p_1 \qquad p_2 = p_1 \sqcup_p (d_2, \mathsf{false}) \qquad p_2 \neq \top_p}{\langle S;\, \texttt{put}\ l\ d_2\rangle \longrightarrow \langle S[l \mapsto p_2];\, ()\rangle} \qquad \text{E-PUT-ERR}\ \frac{S(l) = p_1 \qquad p_1 \sqcup_p (d_2, \mathsf{false}) = \top_p}{\langle S;\, \texttt{put}\ l\ d_2\rangle \longrightarrow \textbf{error}}$$

$$\text{E-GET}\ \frac{S(l) = p_1 \qquad incomp(P) \qquad p_2 \in P \qquad p_2 \sqsubseteq_p p_1}{\langle S;\, \texttt{get}\ l\ P\rangle \longrightarrow \langle S;\, p_2\rangle} \qquad \text{E-FREEZE-INIT}\ \frac{}{\langle S;\, \texttt{freeze}\ l\ \texttt{after}\ Q\ \texttt{with}\ \lambda x.\, e\rangle \longrightarrow \langle S;\, \texttt{freeze}\ l\ \texttt{after}\ Q\ \texttt{with}\ \lambda x.\, e, \{\}, \{\}\rangle}$$

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

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

Figure 2. An operational semantics for LVish.

for parallel composition, which computes two subexpressions e_1 and e_2 in parallel before computing e_3:

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

Although e_1 and e_2 are evaluated in parallel, e_3 cannot be evaluated until both e_1 and e_2 are evaluated, because the call-by-value semantics does not allow β-reduction until the operand is fully evaluated, and because it further disallows reduction under λ-terms (sometimes called "full β-reduction"). In the terminology of concurrent programming, `let par` encompasses both a *fork* and a *join*.

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, for instance, allocating a new LVar before beginning a sequence 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 $(\bot, \mathsf{false})$, and returns the location l of that LVar (*i.e.*, a pointer to the LVar).
- `put` (implemented by the E-PUT and E-PUT-ERR rules) takes a pointer to an LVar and a new lattice element d_2 and updates the LVar's state to the *least upper bound* of the current state and (d_2, false), potentially pushing the state of the LVar upward in the lattice. Any update that would take the state of an LVar to $\top_p$ results in the program immediately stepping to **error**.
- `get` (implemented by the E-GET rule) performs a blocking threshold read. It takes a pointer to an LVar and a *threshold set* P, which is a non-empty set of LVar states that must be *pairwise incompatible*, expressed by the premise $incomp(P)$. A threshold set P is pairwise incompatible iff the lub of any two distinct elements in P is $\top_p$. If the LVar's state p_1 in the lattice is *at or above* some $p_2 \in P$, the `get` operation unblocks and returns p_2. Note that p_2 is a unique element of P, for if there is another $p_2' \neq p_2$ in the threshold set such that $p_2' \sqsubseteq_p p_1$, it would follow that $p_2 \sqcup_p p_2' = p_1 \neq \top_p$, which contradicts the requirement that P be pairwise incompatible.[4]

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 $\texttt{freeze}\ e_{\text{lv}}\ \texttt{after}\ e_{\text{events}}\ \texttt{with}\ e_{\text{cb}}$ primitive, which:

- 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 goes above. The callback e_{cb} is a function that takes a lattice element as its argument. Its return value is ignored, so it runs solely for effect. For instance, a callback might itself do a `put` to the LVar to which it is attached, triggering yet more callbacks.
- If the handler reaches a quiescent state, the LVar e_{lv} is frozen, and its *exact* state is returned (rather than an underapproximation of the state, as with `get`).

To keep track of the 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.

[4] Although $incomp(P)$ is given as a premise of the E-GET reduction rule (suggesting that it is checked at runtime), in a real implementation the incompatibility condition on threshold sets might be checked statically, eliminating the need for the runtime check. In fact, our Haskell implementation (Section 6) has no explicit notion of threshold sets, at runtime or otherwise.

The rule E-SPAWN-HANDLER launches a new callback thread any time the LVar's current value is 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` 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, and our implementation provides a special `runParThenFreeze` function that does so, and guarantees determinism.

4.7 Modeling Lattice Parameterization in Redex

We have developed a runnable model of LVish using the PLT Redex semantics engineering toolkit [10] (included in the non-anonymous supplemental material submitted with this paper). In the Redex of today, it is not possible to directly parameterize a language definition by a lattice. Instead, we define a Racket macro, `define-LVish-language`, that wraps a template implementing the lattice-agnostic semantics of Figure 2. `define-LVish-language` 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 lattice in question. For instance, to instantiate a model called `nat`, where the user-specified 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.

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 non-anonymous supplemental material submitted with this paper.

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 $\sigma \longmapsto^* \sigma'$ and $\sigma \longmapsto^* \sigma''$, 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 $\sigma \equiv \langle S;\, e\rangle \longmapsto \sigma_a$ and $\sigma \longmapsto \sigma_b$, then either:*

1. *there exist π, i, j and σ_c such that $\sigma_a \longmapsto^i \sigma_c$ and $\sigma_b \longmapsto^j \pi(\sigma_c)$ and $i \leq 1$ and $j \leq 1$, or*
2. *$\sigma_a \longmapsto$ **error** or $\sigma_b \longmapsto$ **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 $\langle S;\, e\rangle \longmapsto \langle S';\, e'\rangle$ (where $\langle S';\, e'\rangle \neq$ **error**), then we have that:*
$\langle S \sqcup_S S'';\, e\rangle \longmapsto \langle S' \sqcup_S S'';\, e'\rangle$,
where S'' is any store meeting the following conditions:

- *S'' is non-conflicting with $\langle S;\, e\rangle \longmapsto \langle S';\, e'\rangle$,*
- *$S' \sqcup_S S'' =_{frz} S$, 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 6. 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 7. 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 as part of our supplementary material. Our library adopts the basic approach of the `Par` monad [19], 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.

Embedding the library in Haskell makes it possible to make strong guarantees, because programs written using our library (*i.e.*, that live in our `Par` monad) *only* engage in LVish-sanctioned side-effects. We take full advantage of this fact by indexing our `Par` monad with a phantom type giving its determinism level:

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

The `Par` type constructor has the following kind:[5]

```
Par :: Determinism → * → *
```

together with the following suite of `run` functions:

```
runPar   :: Par Det a → a
runParIO :: Par lvl a → IO a
runParThenFreeze :: (DeepFreeze a b) ⇒ Par Det a → b
```

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 can be executed in the `IO` monad, 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.[6]

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 of the data structure correspond to a lattice, and that the exposed interface to it corresponds to some use of `put`, `get`, 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, this interface might include key/value insertion, lookup, event handlers and pools, and freezing—along with higher-level abstractions built on top of these.

[5] We are here using the "datakinds" extension to Haskell to treat `Determinism` as a kind. In the full implementation, we follow the pattern of the `ST` monad, including another phantom type parameter to ensure that LVars cannot be used in multiple runs of the `Par` monad.

[6] `DeepFreeze` is a typeclass of values of type `a` freezable to type `b`.

For this approach to provide good scaling with available parallelism, 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 large literature on scalable parallel data structures, employing techniques like fine-grained locking and lock-free data structures [12]. 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 [3]. 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 We began with the idea that monotonic data structures add "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 LVish lattices 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 atomic 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 isomorphic to the atoms. 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` have no effect, and since `puts` 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 [20]: 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 [7]). 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.[7] 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. The bag

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

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 scheduling queue. Rather than returning a final result, a completed ClosedPar computation must call the scheduler, sched, on the queue. A Par computation, on the other hand, completes by passing its intended result to its continuation—yielding a ClosedPar computation.

Figure 3 gives the implementation for three core lattice-generic functions: getLV, putLV, and freezeLV.

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 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 checks whether a particular update puts the LVar above some lattice value(s). Both functions return Just r if the threshold has been passed, where r is the result of the read. For 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.

```
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 3. Implementation of key lattice-generic functions

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, which actually 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; while we could require the underlying data structure to support such transactions, doing so would come at a significant price in performance.

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 frozen flag afterward, for a rather subtle reason: suppose the `put` is executing concurrently with a `get` whose threshold it crosses, 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`, but by the time `putLV` reads the frozen 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 `puts` that *caused* the freeze to take place check the frozen status *before* the freeze takes place; additional `puts` that arrive concurrently may, of course, set a mark bit again after `freezeLV` has observed a clear status.

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

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

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

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

where SNZI is a *scalable non-zero indicator* [9], a high-performance parallel counter that supports increment, decrement, and checks for equality with zero. We use the SNZI data structure 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-based 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 is crucial: synchronizing 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) would require maintaining an explicit shared set of handled keys—as expensive as maintaining the map in the first place! That said, data structure authors can implement such synchronization between their threshold functions if desired.

7. Evaluation

We now evaluate the expressiveness and performance of the Haskell LVish implementation. We expect LVish to particularly shine for: parallelizing complicated algorithms on structured data that pose challenges for other deterministic paradigms, and serving as a composition layer for pipeline parallelism between different stages of a computation (each of which may be internally parallelized). Thus we focus on two case studies that fit within this space: control-flow analysis and graph algorithms.

Open benchmarking process While we lack space here to report on all the details of our benchmarks, the non-anonymous supplemental material submitted with this paper contains links to our open benchmarking framework. The logs for all benchmark runs are available in a Jenkins CI instance (click "Build Now" to rerun the benchmarks), and the data set is openly available as a Google Fusion Table, which supports browsing and plotting via a web interface, and live, automatic update with new data.

7.1 Case Study 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 from a version by Might [21].[8] The algorithm processes a continuation-passing-style λ-calculus as its input language. It resembles a non-deterministic abstract interpreter, where stores map addresses onto *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

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

up the abstract state of the program, and the goal is to explore a graph of these states.

The heart of the search process is the following function:

```
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 datatypes like `Data.Set` and lists. Some performance problems are immediately apparent. For example, the set of next states is converted to a list so that it can be streamed in order through the `todos`. From our standpoint, the problem is a deep one: in a purely functional program with a balanced-tree representation of sets, there is no good way to compute new states and add them to the `seen` set in parallel! Even if one computes the neighbor states in parallel, adding them to book-keeping structures (*e.g.*, `seen`) effectively serializes the computation before proceeding any further in the search. We attempted to parallelize at the algorithm point using traditional, purely functional parallelism (futures/`parBuffer`), but were not able to achieve any speedup.

Worse yet, the purely functional implementation creates many diverging copies of the store, only to union them back together:

```
summarize states = fold (λ(State _ _ store' _) store →
                           store `storeJoin` store')
                        empty states
```

Avoiding this problem would require invasive changes to the program and would still not present opportunities for parallelism.

Porting to the LVish library We began by doing a verbatim port to LVish. A purely functional program is a good place to start, because it is possible to allocate a new LVar for each new value in the original program, with standard `map` and `fold`-style operations making it straightforward to write LVish programs in idiomatic style. This is done simply by refactoring two types (`Set` and `Map`) into their monotonic LVar counterparts, `ISet` and `IMap`:

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

The `explore` function from above can then be replaced by the simple graph traversal function from Section 1! The rest of the changes to the program are mechanical, including converting pure to monadic code. The key for the programmer is to consume LVars as if they were pure values, ignoring the fact that an LVar's contents are spread out over space and time and are modified through effects.

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

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

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

Flipping the switch The verbatim LVish port has a small performance overhead relative to the original, but is a much easier starting point for overcoming the performance problems mentioned above. For example, the `next` function for our port, like the original code, creates a fresh store to extend with new bindings as we explore a new part of the state space:

```
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 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).

Figure 4 shows performance data for the "blur" benchmark drawn from a recent paper on k-CFA [8]. (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. We also created one synthetic benchmark that manages 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), shown in Figure 4. It has a small state space of large states with many variables (600 states / 1211 variables).

The role of lock-free data structures As part of our library, we provide a lock-free implementation of finite maps and sets based on a concurrent skip list [12].[9] We also provide reference implementations that use a nondestructive `Data.Set` inside a mutable container. Our scalable implementation is not yet carefully optimized, and at one and two cores lock-free k-CFA is 38% to 43% slower than the reference implementation. But the effect of scalable data structures is quite visible on a 12-core machine.[10] Without them, the blur benchmark (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 the `ST` monad allows access to efficient in-place array computations.

7.2 Case Study 2: Graph Algorithms

The recently proposed *Problem Based Benchmark Suite* [4] contains numerous benchmarks designed to focus on non-numerical computing on irregular problems. We have implemented and measured two of the benchmarks from this suite: *breadth-first search* and *maximal independent set* benchmarks (BFS and MIS). The full results are given in the non-anonymous supplemental material.

In short, we observe between $2.09\times$ and $2.76\times$ parallel speedup on raw BFS (with no other handlers registered), depending on topology, and $1.34 - 1.79\times$ on MIS. These problems are very fine-grained. Yet when BFS or MIS are used to discover a part of the graph that is used as input to nontrivial downstream computations (*e.g.*, $1 - 25\mu s$ of work per vertex), scaling improves and the

[9] In fact, this project is the first to incorporate *any* lock-free data structures in Haskell, which required solving some unique problems pertaining to Haskell's lazy implementation and the GHC compiler which assumes referential transparency. But we lack the space to detail these improvements.

[10] Intel Xeon 5660; full machine details here `https://portal.futuregrid.org/hardware/delta`.

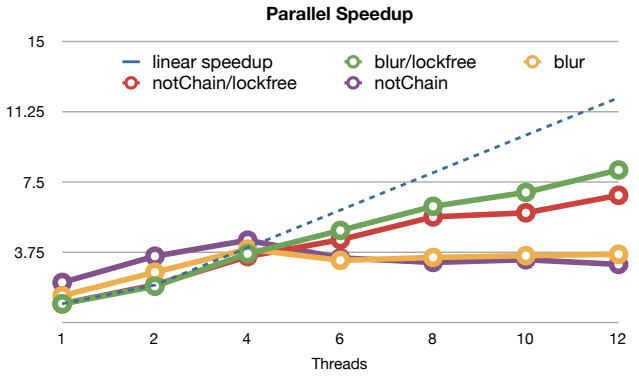

Figure 4. Time to analyze the "blur" and "notChain" benchmarks. Parallel speedup is normalized to the sequential times for the *lock-free* versions (5.21s and 9.83s, respectively). The normalized speedups are remarkably consistent for the lock-free version between the two benchmarks. But the relationship to the original, purely functional version is quite different: at 12 cores, the lock-free LVish version of "blur" is 202× faster than the original, while "notChain" is only 1.6× faster, not gaining anything from sharing rather than copying stores due to a lack of fan-out in the state graph.

benefits of LVish become apparent in enabling downstream phases to start sooner. For example, if a BFS stage feeds into a per-vertex computation, but the graph turns out to have long linear chains, the PBBS C++/Cilk implementation is slowed down to roughly the speed of LVish/Haskell *and* subsequent phases are blocked. For example, with a 10M vertex chain, subsequent tasks must wait 9 seconds to start vs. 0.14 to 0.23 milliseconds with LVish (irrespective of the number of threads).

8. Discussion and Related Work

As mentioned in Section 1, what deterministic parallel models have in common is that they all must do something to tame shared mutable state—whether by disallowing sharing entirely [13], only allowing single assignments to shared references [2, 6, 26], allowing sharing only by a limited form of message passing [14], ensuring that concurrent accesses to shared state are disjoint [5], or with some combination of these approaches. These constraints can be imposed at the language/API level, within a type system, or at runtime. LVish is an attempt to extend the reach of language-level (quasi-)determinism guarantees, by supporting both a wide variety of scalable parallel data structures and a rich, asynchronous programming model. In addition to the models cited throughout the paper, here we consider the most closely related recent work.

Prokopec *et al*. [23] propose *FlowPools*, 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 basic interface has a similar flavor 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, LVish provides the more general `freeze` operation, which is more expressive and convenient, but moves the design into the realm of quasi-determinism. Another important difference is the fact that LVish is *data structure-generic*: our theorems and our library support an unlimited collection of data structures, whereas FlowPools included a bag-specific determinism proof and implementation.

The Concurrent Revisions (CR) [18] programming model uses isolation types to distinguish regions of the heap shared by multiple mutators. Rather than enforcing exclusive access, CR clones a copy of the state for each mutator, using a deterministic policy for resolving conflicts in local copies at join points. 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.

In the distributed systems literature, *eventually consistent* systems based on *conflict-free replicated data types* [24] leverage the idea of lattice monotonicity to guarantee that replicas in a distributed database eventually agree. For instance, the Bloom language for distributed database programming [1] guarantees eventual consistency for distributed data collections that are updated monotonically. This process is analogous to quiescence—a transient property, as we have noted—and such systems make no guarantees about determinism: as such, writes are least-upper-bound writes like our `put`, but direct reads are allowed.

References

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

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

[3] 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.

[4] G. E. Blelloch, J. T. Fineman, P. B. Gibbons, and J. Shun. Internally deterministic parallel algorithms can be fast. In *PPoPP*, 2012.

[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] 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.

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

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

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

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

[11] 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.

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

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

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

[15] J. B. Kam and J. D. Ullman. Monotone data flow analysis frameworks. *Acta Informatica*, 7(3):305–317, Sept. 1977.

[16] L. Kuper and R. R. Newton. LVars: lattice-based data structures for deterministic parallelism. In *FHPC*, 2013. URL `https://www.cs.indiana.edu/~lkuper/papers/2013-lvars-draft.pdf`. To appear.

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

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

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

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

[21] 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/.

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

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

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

[25] J. Shun, G. E. Blelloch, J. T. Fineman, P. B. Gibbons, A. Kyrola, H. V. Simhadri, and K. Tangwongsan. Brief announcement: the Problem Based Benchmark Suite. In *SPAA*, 2012.

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