LVars: Lattice-based Data Structures for Deterministic Parallelism

Lindsey Kuper     Ryan R. Newton
Indiana University
{lkupe, rrenwot} @cs.indiana.edu

Abstract

Programs written using a deterministic-by-construction model of parallel computation are guaranteed to always produce the same observable results, offering programmers freedom from subtle, hard-to-reproduce nondeterministic bugs that are the scourge of parallel software. We present LVars, a new model for deterministic-by-construction parallel programming that generalizes existing single-assignment models to allow multiple assignments that are monotonically increasing with respect to a user-specified lattice. LVars ensure determinism by allowing only monotonic writes and “threshold” reads that block until a lower bound is reached. We give a proof of determinism and a prototype implementation for a language with LVars and describe how to extend the LVars model to support a limited form of nondeterminism that admits failures but never wrong answers.

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

1. Introduction

Programs written using a deterministic-by-construction model of parallel computation are guaranteed to always produce the same observable results, offering programmers freedom from subtle, hard-to-reproduce nondeterministic bugs that are the scourge of parallel software. While a number of popular languages and language extensions (e.g., Cilk [14]) encourage deterministic parallel programming, few of them guarantee determinism for all programs written using the model.

The most developed parallel model that offers a deterministic-by-construction guarantee for all programs—“developed” here meaning mature implementations, broadly available, with many libraries and reasonable performance—is pure functional programming with function-level task parallelism, or futures. For example, Haskell programs using futures by means of the par and parseq combinators can provide real speedups on practical programs while guaranteeing determinism [21]. Yet pure programming with futures is not ideal for all problems. Consider a producer/consumer computation in which producers and consumers can be scheduled onto separate processors, each able to keep their working sets in cache. Such a scenario enables pipeline parallelism and is common, for instance, in stream processing. But a clear separation of producers and consumers is difficult with futures, because whenever a consumer forces a future, if it is not yet available, the consumer immediately switches roles to begin computing the value (as explored in previous work [23]).

Since pure programming with futures is a poor fit for producer/consumer computations, one might then turn to stateful deterministic parallel models. Shared state between computations allows the possibility for data races that introduce nondeterminism, so any parallel model that hopes to preserve determinism must do something to tame sharing—that is, to restrict access to mutable state shared among concurrent computations. Systems such as DPJ (Deterministic Parallel Java) [6] and Concurrent Revisions [8, 19], for instance, accomplish this by ensuring that the state accessed by concurrent threads is disjoint.

In this paper, we are concerned with an alternative approach: allowing data to be shared, but limiting the operations that can be performed on it to only those operations that commute with one another and thus can tolerate nondeterministic thread interleavings. Although the order in which side-effecting operations occur can differ on multiple runs, a program will always produce the same externally observable result [1]. Specifically, we are concerned with models where shared data structures grow monotonically—by publishing information, but never invalidating it. These models support pipelining for producer/consumer applications.

Existing monotonie models Consider two classic deterministic parallel models, dating back to the late 60s and early 70s [16, 27]:

- In Kahn process networks (KPNs) [16], as well as in the more restricted synchronous data flow systems [13], a network of processes communicate with each other through blocking FIFO channels. KPNs are the basis for deterministic stream-processing languages such as StreamIt [15], which are narrowly focused but have shown clear benefits in auto-parallelization and hardware portability.

- In parallel single-assignment languages [27], “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 are known as LVars [4] and are a well-established

1 With SafeHaskell enabled and, of course, no IO.
2 There are many ways to define what is observable about a program. In this paper, we define the observable result of a program to be the value to which it evaluates.
mechanism for enforcing determinism in parallel settings: they have appeared in Concurrent ML as SyncVars [24]; in the Intel Concurrent Collections (CnC) system [7], in languages and libraries for high-performance computing, such as Chapel [9] and the Othreads library [29]; and have even been implemented in hardware in Cray MTA machines [5]. Although most of these uses incorporate Ivars into already-nondeterministic programming environments, the monad-par Haskell library [22] uses Ivars in a deterministic-by-construction setting, allowing user-created threads to communicate through Ivars without requiring 10, so that such communication can occur anywhere inside pure programs.

In data-flow languages like StreamIt, communication takes place over FIFOs with ever-increasing channel histories, while in Ivar-based systems such as CnC and monad-par, a shared data store of single-assignment memory cells grows monotonically. Hence monotonic data structures—those to which information is only added and never removed—emerge as a common theme of both data-flow and single-assignment models.

Because state modifications that only add information and never destroy it can be structured to commute with one another and thereby avoid race conditions, it stands to reason that diverse deterministic parallel programming models would leverage the principle of monotonicity. Yet there is little in the way of a theory of monotonic data structures as a basis for deterministic parallelism. As a result, systems like CnC, monad-par and StreamIt emerge independently, without recognition of their common basis. Moreover, they lack generality: Ivars and FIFO streams alone cannot support all producer/consumer applications, as we discuss in Section 5.

A general model By taking monotonicity as a starting point, then, we can provide a new model for deterministic parallelism that generalizes existing models and can guide the design of new ones. Our model generalizes Ivars to Lvars, thus named because the states an LVar can take on are elements of a user-specified lattice. This user-specified lattice determines the semantics of the put and get operations that comprise the interface to Lvars (which we will explain in detail in Section 3.3).

• The put operation can only change an LVar’s state in a way that is monotonically increasing with respect to the user-specified lattice, because it takes the least upper bound of the current state and the new state.

• The get operation allows only limited observations of the state of an LVar. It requires the user to specify a threshold set of minimum values that can be read from the LVar, where every two elements in the threshold set must have the lattice’s greatest element ⊤ as their least upper bound. A call to get blocks until the LVar in question reaches a (unique) value in the threshold set, then unblocks and returns that value.

Together, monotonically increasing writes via put and threshold reads via get yield a deterministic-by-construction programming model. We use LVars to define λLVar, a deterministic parallel calculus with shared state, based on the call-by-value λ-calculus. The λLVar language is general enough to subsume existing deterministic parallel languages because it is parameterized by the choice of lattice. For example, a lattice of channel histories with a prefix order allows LVars to represent FIFO channels that implement a Kahn process network, whereas instantiating λLVar with a lattice with “empty” and “full” states (where empty < full) results in a parallel single-assignment language. Different instantiations of the lattice result in a family of deterministic parallel languages.

Because lattices are composable, any number of diverse monotonic data structures can be used together safely. Moreover, as long as we can demonstrate that a data structure presents the LVar interface, it is fine to use an existing, optimized concurrent data structure implementation; we need not rewrite the world’s data structures to leverage the λLVar determinism result.

Contributions

• We introduce LVars (Section 3) and use them to define λLVar, a parallel calculus that uses LVars for shared state (Section 4). We have implemented λLVar as a runnable PLT Redex model.

• As our main technical result, we give a proof of determinism for λLVar (Section 5). A critical aspect of the proof is a “frame” property, expressed by the Independence lemma (Section 5.1), that would not hold in a typical language with shared mutable state, but holds in our setting because of the semantics of LVars and their put/get interface.

• We describe an extension to the basic λLVar model that allows destructive observations of LVars by means of a consume operation, enabling a limited form of nondeterminism that admits run-time failures but not wrong answers (Section 7), and we give an implementation of a data-race detector that detects such failures in a version of λLVar, that has been extended with consume.

• We discuss how to formulate common data structures (pairs, trees, arrays, FIFOs) as lattices (Sections 3.1 and 4.2) and how to implement operations on them within the λLVar model—for instance, we show how to implement a bump operation that increments a monotonic counter represented as a lattice (Section 7.1).

• We provide a practical prototype implementation of LVars as an extension to the monad-par Haskell library, and give some preliminary benchmarking results (Section 9).

All the code accompanying this paper is available at https://github.com/iu-parfunc/lvars which we refer to as “the LVars repository” throughout the paper.

2. Motivating Example: A Parallel, Pipelined Graph Computation

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

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

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

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

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

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

Unfortunately, the code given in Figure 1 does not quite implement the problem specification given above. Even though connected-component discovery is parallel, members of the output set do not become available to other computations until component discovery is finished, limiting parallelism. We could manually push the analyze invocation inside the \( \text{bf\_traverse} \) function, allowing the analyze computation to start sooner, but then we push the same problem to the downstream consumer, unless we are able to perform a heroic whole-program fusion. If \( \text{bf\_traverse} \) returned a list, lazy evaluation could make it possible to stream results to consumers incrementally. But with a set result, such pipelining is not generally possible: consuming the results early would create a proof obligation that the determinism of the consumer does not depend on the order in which results emerge from the producer.

A compromise would be for \( \text{bf\_traverse} \) to return a list of level-sets: distance one nodes, distance two nodes, and so on. Thus level-one results could be consumed before level-two results are ready. Still, the problem would remain: within each level-set, we cannot launch all instances of \( \text{analyze} \) and asynchronously use those results that finish first. Furthermore, we still have to contend with the previously-mentioned difficulty of separating producers and consumers when expressing producer-consumer computations using pure programming with futures [22].

Our solution Suppose that we could write a breadth-first traversal in a programming model with limited effects that allows any shared data structure between threads, including sets and graphs, so long as that data structure grows monotonically. Consumers of the data structure may execute as soon as data is available, but may only observe irrevocable, monotonic properties of it. This is possible with a programming model based on LVars. After introducing the \( \lambda_{LVar} \) calculus and giving its determinism proof in the next few

---

\( \text{nbrs} :: \text{Graph} \to \text{NodeLabel} \to \text{Set NodeLabel} \)

-- 'nbrs g n' is the neighbor nodes of 'n' in 'g'

-- Traverse each level of the graph in parallel,
-- maintaining at each recursive step a set of
-- nodes that have been seen and a set of nodes
-- left to process.

\( \text{bf\_traverse} :: \text{Graph} \to \text{Set NodeLabel} \to \)

\( \text{Set NodeLabel} \to \text{Set NodeLabel} \)

\( \text{bf\_traverse} g \text{ seen } n u = \)

\( \text{if } n u = \{\} \)

\( \text{then } \text{seen } \) = union \( \text{seen } n u \)

\( \text{allNbr} = \text{parFold union} \) (parMap (nbrs g) nu)

\( \text{nu } = \text{difference} \) allNbr \( \text{seen }' \)

\( \text{in } \text{bf\_traverse} \ g \ \text{seen }' \ n u' \)

-- Next we traverse the connected component,
-- starting with the vertex 'profile0':

\( \text{ccmp } = \text{bf\_traverse} \ \text{profiles} \{\} \) \{profile0\}

\( \text{result } = \text{parMap analyze ccmp} \)

---

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

---

Figure 2. Example lattices: (a) IVar containing a natural number; (b) pair of natural-number-valued IVars; (c) positive integers ordered by \( \leq \) (see Section 2). Subfigure (b) is annotated with example threshold sets that would correspond to a blocking read of the first or second element of the pair (see Sections 3.1 and 2.1). Any state transition crossing the “tripwire” for \( \text{getFst} \) causes it to unblock and return a result.

---

3. Lattices, Stores, and Determinism

As a minimal substrate for LVars, we introduce \( \lambda_{LVar} \), a parallel call-by-value \( \lambda \)-calculus extended with a \text{store} and with communication primitives \text{put} and \text{get} that operate on data in the store. The class of programs that we are interested in modeling with \( \lambda_{LVar} \) are those with explicit effectful operations on shared data structures, in which subcomputations may communicate with each other via the \text{put} and \text{get} operations.

In this setting of shared mutable state, the trick that \( \lambda_{LVar} \) employs to maintain determinism is that stores contain \( LVars \), which are a generalization of IVars [4]—namely, those with only one cell.

---

5 As intuition for this idea, consider that purely functional set data structures, such as Haskell’s \text{Data.Set}, are typically represented with balanced trees. Unlike with lists, the structure of the tree is not known until all elements are present.

6 IVars are so named because they are a special case of I-structures [4]—namely, those with only one cell.
variables—either empty or filled with an immutable value—an LVar may have an arbitrary number of states forming a set D, which is partially ordered by a relation ⊑. An LVar can take on any sequence of states from D, so long as that sequence respects the partial order—that is, updates to the LVar (made via the put operation) are inflationary with respect to ⊑. Moreover, the get operation allows only limited observations of the LVar’s state. In this section, we discuss how lattices and stores work in λLVar and explain how the semantics of put and get together enforce determinism in λLVar programs.

3.1 Lattices

The definition of λLVar is parameterized by the choice of D: to write concrete λLVar programs, one must specify the set of LVar states that one is interested in working with, and an ordering on those states. Therefore λLVar is actually a family of languages, rather than a single language. Formally, D is a bounded join-semilattice augmented with a greatest element ⊤.

• D has a least element ⊥, representing the initial “empty” state of an LVar.

• D has a greatest element ⊤, representing the “error” state that results from conflicting updates to an LVar.

• D comes equipped with a partial order ⊑, where ⊥ ⊑ d ⊑ ⊤ for all d ∈ D.

• Every pair of elements in D has a least upper bound (lub) ⊔. Intuitively, the existence of a lub for every two elements in D means that it is possible for two subcomputations to independently update an LVar, and then deterministically merge the results by taking the lub of the resulting two states.

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

Figure 2 gives three examples of lattices for common data structures. The simplest example of a useful lattice is one that represents the state space of a single-assignment variable (an IVar). A natural-number-valued IVar, for instance, would correspond to the lattice in Figure 2(a), that is,

\[ D = (\{\bot, \top\} \cup \mathbb{N}, \subseteq) \]

where the partial order ⊑ is defined by setting ⊥ ⊑ d ⊑ ⊤ and d ⊑ d for all d ∈ D. This is a lattice of height three and infinite width, where the naturals are arranged horizontally. After the initial write of some n ∈ \mathbb{N}, any further conflicting writes would push the state of the IVar to \top (an error). For instance, if one thread writes 2 and another writes 1 to an IVar (in arbitrary order), the second of the two writes would result in an error because 2 ⊑ 1 = ⊤.

In the lattice of Figure 2(b), on the other hand, the \top state can never be reached, because the least upper bound of any two writes is just the maximum of the two. For instance, if one thread writes 2 and another writes 1 to an IVar (in arbitrary order), the second of the two writes would result in an error because 2 ⊒ 1 = ⊤. Here, the unreachability of \top models the fact that no conflicting updates can occur to the LVar.

3.2 Stores

During the evaluation of a λLVar program, a store S keeps track of the states of LVars. Each LVar is represented by a binding from a location l, drawn from a set Loc, to its state, which is some element d ∈ D. Although each LVar in a program has its own state, the states of all the LVars are drawn from the same lattice D. We can do this with no loss of generality because lattices corresponding to different types of LVars could always be unioned into a single lattice (with shared \top and ⊥ elements). Alternatively, in a typed formulation of λLVar, the type of an LVar might determine the lattice of its states.

Definition 1. A store is either a finite partial mapping \( S : \text{Loc} \rightarrow (D - \{\top\}) \), or the distinguished element \( \bot_S \).

We use the notation S[l → d] to denote extending S with a binding from l to d. If l ∈ dom(S), then S[l → d] denotes an update to the existing binding for l, rather than an extension. We can also denote a store by explicitly writing out all its bindings, using the notation \([l_1 \mapsto d_1, \ldots, l_n \mapsto d_n]\). The state space of stores forms a bounded join-semilattice augmented with a greatest element, just as D does, with the empty store \( \bot_S \) as its least element and \( \top_S \) as its greatest element. It is straightforward to lift the ⊑ and ⊔ operations defined on elements of D to the level of stores.

Definition 2. A store S is less than or equal to a store S′ (written \( S \subseteq_S S' \)) if:

• \( S' = \top_S \), or

• \( \text{dom}(S) \subseteq \text{dom}(S') \) and for all \( l \in \text{dom}(S) \), \( S(l) \subseteq S'(l) \).

Definition 3. The least upper bound (lub) of two stores S1 and S2 (written \( S_1 \sqcup_S S_2 \)) is defined as follows:

\[ S'(l) = \begin{cases} S_1(l) \sqcup_S S_2(l) & \text{if } l \in \text{dom}(S_1) \cap \text{dom}(S_2) \\ S_1(l) & \text{if } l \notin \text{dom}(S_2) \\ S_2(l) & \text{if } l \notin \text{dom}(S_1) \end{cases} \]

By Definition 3, if \( d_1 \sqcup_S d_2 = \top \), then \([l \mapsto d_1] \cup_S [l \mapsto d_2] = \top_S \). Notice that a store containing a binding \( l \mapsto \top \) can never arise during the execution of a λLVar program, because (as we will see in Section 3.3) an attempted write that would take the state of l to \top would raise an error before the write can occur.

3.3 Communication Primitives

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

• new extends the store with a binding for a new LVar whose initial state is \⊥, and returns the location l of that LVar (i.e., a pointer to the LVar).

• put takes a pointer to an LVar and a singleton set containing a new state and updates the LVar’s state to the least upper bound of the current state and the new state, potentially pushing the state of the LVar upward in the lattice. Any update that would take the state of an LVar to \top results in an error.

• get performs a blocking “threshold” read that allows limited observations of the state of an LVar. It takes a pointer to an LVar and a threshold set Q, which is a non-empty subset of D that is pairwise incompatible, meaning that the lub of any two distinct elements in Q is \top. If the LVar’s state d in the lattice is at or above some \( d' \in Q \), the get operation unblocks and returns the singleton set \{d′\}. Note that \( d' \) is a unique element of Q, for if there is another \( d'' \neq d' \) in the threshold set such that \( d'' \subseteq d \), it would follow that \( d' \sqcup d'' \subseteq d \neq \top \), which contradicts the requirement that Q be pairwise incompatible.
The intuition behind get is that it specifies a subset of the lattice that is “horizontal”: no two elements in the threshold set can be above or below one another. Intuitively, each element in the threshold set is an “alarm” that detects the activation of itself or any state above it. One way of visualizing the threshold set for a get operation is as a subset of edges in the lattice that, if crossed, set off the corresponding alarm. Together these edges form a “trip-wire”. This visualization is pictured in Figure 2(b). The threshold set \( \{ (\bot, 0), (\bot, 1), \ldots \} \) (or a subset thereof) would pass the incompatibility test, as would the threshold set \( \{ (0, \bot), (1, \bot), \ldots \} \) (or a subset thereof), but a combination of the two would not pass.

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

3.4 Monotonic Store Growth and Determinism

In IVar-based languages, a store can only change in one of two ways: a new binding is added at \( \bot \), or a previously \( \bot \) binding is permanently updated to a meaningful value. It is therefore straightforward in such languages to define an ordering on stores and establish determinism based on the fact that stores grow monotonically with respect to the ordering. For instance, Featherweight CnC [7], a single-assignment imperative calculus that models the Intel Concurrent Collections (CnC) system, defines ordering on stores as follows:

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

Our Definition 2 is reminiscent of Definition 4, but Definition 4 requires that \( S(l) \) and \( S'(l) \) be equal, instead of our weaker requirement that \( S(l) \subseteq S'(l) \) according to the user-provided partial order \( \preceq \). In \( \lambda_{\text{LVar}} \), stores may grow by updating existing bindings via repeated puts, so Definition 4 would be too strong; for instance, if \( \bot \preceq d_1 \preceq d_2 \) for distinct \( d_1, d_2 \in D \), the relationship \( l \mapsto d_1 \subseteq S \) \( l \mapsto d_2 \) holds under Definition 4, but would not hold under Definition 2. That is, in \( \lambda_{\text{LVar}} \) an LVar could take on the state \( d_1 \) followed by \( d_2 \), which would not be possible in Featherweight CnC. We establish in Section 5 that \( \lambda_{\text{LVar}} \) remains deterministic despite the relatively weak \( \subseteq \) relation given in Definition 2. The keys to maintaining determinism are the blocking semantics of the get operation and the fact that it allows only limited observations of the state of an LVar.

4. \( \lambda_{\text{LVar}} \): Syntax and Semantics

The syntax and operational semantics of \( \lambda_{\text{LVar}} \) appear in Figures 3 and 4, respectively. As we have noted, both the syntax and semantics are parameterized by the lattice \( D \). The reduction relation \( \beta \) 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 \( \langle \bot_S; e \rangle \) to be equal to error, for all expressions \( e \). The metavariable \( \sigma \) ranges over configurations.

Figure 5 shows two disjoint sets of reduction rules: those that step to configurations other than error, and those that step to error. Most of the latter set of rules merely propagate errors along. A new error can only arise by way of the E-PAPPERR rule, which represents the joining of two conflicting subcomputations, or by

Given a lattice \( (D, \subseteq) \) with elements \( d \in D \), least element \( \bot \), and greatest element \( \top \):

- **configurations** \( \sigma \) ::= \( \langle S; e \rangle \) | error
- **expressions** \( e \) ::= \( x \) | \( v \) | error | new | put \( e \) | convert \( e \) | get \( e \)
- **values** \( v \) ::= \( l \) | \( Q \) | \( \lambda x. e \)
- **threshold set literals** \( Q \) ::= \{ \( d_1, d_2, \ldots \) \}
- **stores** \( S \) ::= \( \top_S \) | \( l_1 \mapsto d_1, \ldots, l_n \mapsto d_n \) (where \( d_i \neq \top \))

**Figure 3.** Syntax for \( \lambda_{\text{LVar}} \).

way of the E-PUTVALERR rule, which applies when a put to a location would take its state to \( \top \).

The rules E-NEW, E-PUTVAL/E-PUTVAL, and E-GETVAL respectively express the semantics of the new, put, and get operations described in Section 3.3. The incompatibility property of the threshold set argument to get is enforced in the E-GETVAL rule by the \( \text{incomp}(Q) \) premise, which requires that the least upper bound of any two distinct elements in \( Q \) must be \( \top \).

The E-PUT-1/E-PUT-2 and E-GET-1/E-GET-2 rules allow for reduction of subexpressions inside put and get expressions until their arguments have been evaluated, at which time the E-PUTVAL (or E-PUTVALERR) and E-GETVAL rules respectively apply. Arguments to put and get are evaluated in arbitrary order, although not simultaneously, for simplicity’s sake. However, it would be straightforward to add E-PARPUT and E-PARGET rules to the semantics that are analogous to E-PARAPP, should simultaneous evaluation of put and get arguments be desired.

4.1 Fork-Join Parallelism

\( \lambda_{\text{LVar}} \) has an explicitly parallel reduction semantics: the E-PARAPP rule in Figure 6 allows simultaneous reduction of the operator and operand in an application expression, so that (eliding stores) the application \( e_1 \ e_2 \) may step to \( e'_1 \ e'_2 \) if \( e_1 \) steps to \( e'_1 \) and \( e_2 \) steps to \( e'_2 \). In the case where one of the subexpressions is already a value or is otherwise unable to step (for instance, if it is a blocked get), the reflexive E-REFL rule comes in handy: it allows the E-PARAPP rule to apply nevertheless. When the configuration \( \langle S; e_1 \ e_2 \rangle \) takes a step, \( e_1 \) and \( e_2 \) step as separate subcomputations, each beginning with its own copy of the store \( S \). Each subcomputation can update \( S \) independently, and we combine the resulting two stores by taking their least upper bound when the subcomputations rejoin. (Because E-PARAPP and E-PARAPPERR perform truly simultaneous reduction, they have to address the subtle point of location renaming: locations created while \( e_1 \) steps must be renamed to avoid name conflicts with locations created while \( e_2 \) steps. We discuss the rename metafunction and other issues related to renaming in Appendix A.)

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

\[
\text{let par } x = e_1 \\
\text{ y = e}_2 \\
\overset{\Delta}{=} ((\lambda x. (\lambda y. e_3)) e_1) e_2 \\
\text{ in } e_3
\]

Although \( e_1 \) and \( e_2 \) can be evaluated in parallel, \( e_3 \) cannot be evaluated until both \( e_1 \) and \( e_2 \) are values, because the call-by-value semantics does not allow \( \beta \)-reduction until the operand is fully evaluated, and because it further disallows reduction under \( \lambda \)-terms (sometimes called “full \( \beta \)-reduction”). In the terminology
Given a lattice \((D, \sqsubseteq)\) with elements \(d \in D\), least element \(\bot\), and greatest element \(\top\):

\[\text{incomp}(Q) \equiv \forall a, b \in Q, (a \neq b \implies a \sqcup b = \top)\]

\[
\begin{align*}
\text{E-REFL} & : \quad (S; e) \rightarrow (S; e) \\
\text{E-PARAPP} & : \quad (S; e_1) \rightarrow (S; e_1') \\
& \quad \rightarrow (S; e_2) \rightarrow (S; e_2') \\
& \quad \rightarrow (S; e_1 \circ e_2) \rightarrow (S; e_1' \circ e_2') \\
\text{E-PARAPP-ERR} & : \quad (S; e_1) \rightarrow \text{error} \\
& \quad \rightarrow (S; e_2) \rightarrow \text{error} \\
\text{E-PUT} & : \quad \text{put} \ e \rightarrow (S; \text{put} \ e) \\
\text{E-PUT-1} & : \quad (S; e_1) \rightarrow (S; e_1') \\
& \quad \rightarrow (S; e_2) \rightarrow (S; e_2') \\
\text{E-PUT-2} & : \quad \text{put} \ e_1 \rightarrow (S; \text{put} \ e_1) \\
& \quad \rightarrow (S; \text{put} \ e_2) \rightarrow (S; \text{put} \ e_2') \\
\text{E-PUT-ERR} & : \quad (S; e_1) \rightarrow \text{error} \\
\text{E-PUT-ERR-1} & : \quad \text{put} \ e \rightarrow \text{error} \\
\text{E-GET} & : \quad \text{get} \ e \rightarrow (S; \text{get} \ e) \\
\text{E-GET-1} & : \quad (S; e_1) \rightarrow (S; e_1') \\
& \quad \rightarrow (S; e_2) \rightarrow (S; e_2') \\
\text{E-GET-2} & : \quad \text{get} \ e_1 \rightarrow (S; \text{get} \ e_1) \\
& \quad \rightarrow (S; \text{get} \ e_2) \rightarrow (S; \text{get} \ e_2') \\
\text{E-GET-ERR} & : \quad (S; e_1) \rightarrow \text{error} \\
\text{E-GET-ERR-1} & : \quad \text{get} \ e \rightarrow \text{error} \\
\text{E-CONVERT} & : \quad (S; e) \rightarrow (S'; e') \\
\text{E-CONVERT-1} & : \quad (S; e) \rightarrow \text{convert} (S; e) \\
\text{E-CONVERT-ERR} & : \quad (S; e) \rightarrow \text{error} \\
\end{align*}
\]

Figure 4. An operational semantics for \(\lambda_{\text{Var}}\).

of parallel programming, a \texttt{let par} expression executes both a \texttt{fork} and a \texttt{join}. Indeed, it is common for \texttt{fork} and \texttt{join} to be combined in a single language construct, for example, in languages with parallel tuple expressions such as Manticore \[13\].

Since \texttt{let par} expresses \texttt{fork-join} parallelism, the evaluation of a program comprising nested \texttt{let par} expressions would induce a runtime dependence graph like that pictured in Figure 5(a). The \(\lambda_{\text{Var}}\) language (minus \texttt{put} and \texttt{get}) can support any \textit{series-parallel} dependence graph. Adding communication through \texttt{put} and \texttt{get} introduces “lateral” edges between branches of a parallel computation, as in Figure 5(b). This adds the ability to construct arbitrary non-series-parallel dependency graphs, just as with \textit{first-class futures} \[26\].

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

4.2 Programming with put and get

For our first example of a \(\lambda_{\text{Val}}\) program, we choose the elements of our lattice to be pairs of natural-number-valued LVars, as shown in Figure 2(b). We can then write the following program:

\begin{verbatim}
let p = new in
let v1 = put p \{(3, 4)\} in
(Example 1)
\end{verbatim}

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

Since threshold sets can be cumbersome to read, we can define some convenient shorthands \texttt{getMax} and \texttt{getMin} for working with
our lattice of pairs:

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

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

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

**Querying incomplete data structures** It is worth noting that getSnd \( p \) returns a value even if the first entry of \( p \) is not filled in. For example, if the put in the second line of (Example 1) had been put \( p \{ (\bot, 4) \} \), the get expression would still return \( \{ (\bot, 4) \} \). It is therefore possible to safely query an incomplete data structure—say, an object that is in the process of being initialized by a constructor. However, notice that we cannot define a \( \text{getFst} \& \text{getSnd} \) function that returns if either entry of a pair is filled in. Doing so would amount to passing all of the boxed elements of the lattice in Figure 2(b) to get as a single threshold set, which would fail to satisfy the incompatibility criterion.

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

\[
\text{let } p = \text{new in}
\]

\[
\text{let } _1 = \text{put } p \{ (\bot, 4) \} \text{ in}
\]

\[
\text{let } \_1 = \text{getFst } p \quad \text{(Example 2)}
\]

\[
\_1 = \text{put } p \{ (3, 4) \}
\]

In... 

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

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

### 4.3 Converting from Threshold Sets to \( \lambda \)-terms and Back

There are two worlds that \( \lambda_{\text{LVar}} \) values may inhabit: the world of threshold sets, and the world of \( \lambda \)-terms. But if these worlds are disjoint—if threshold set values are opaque atoms—certain programs are impossible to write. For example, implementing single-assignment arrays in \( \lambda_{\text{LVar}} \) requires that arbitrary array indices can be computed (e.g., as Church numerals) and converted to threshold sets. Thus we parameterize our semantics by a conversion function, \( \delta : v \rightarrow v \), exposing through the convert language form. The convert function is arbitrary, without any particular structure, similar to including an abstract set of primitive functions in the language. It is optional in the sense that providing an identity or empty function is acceptable, and leaves \( \lambda_{\text{LVar}} \) sensible but less expressive. This is mainly a notational concern that does not have significant implications for a real implementation.

### 5. Proof of Determinism for \( \lambda_{\text{LVar}} \)

Our main technical result is a proof of determinism for the \( \lambda_{\text{LVar}} \) language. Most proofs are only sketched here; the complete proofs appear in the companion technical report [17].

Frame rule (O’Hearn et al., 2001):

\[
\frac{\{ p \} \ e \ \{ q \}}{\{ p \ast r \} \ e \ \{ q \ast r \}} \quad (\text{where no free variable in } r \text{ is modified by } e)
\]

**Lemma 1** (Independence), simplified:

\[
\langle S; e \rangle \rightsquigarrow \langle S'; e' \rangle \quad (S'' \text{ non-conflicting with } \langle S \cup S''; e \rangle \rightarrow \langle S'; e' \rangle)
\]

Figure 7. Comparison of a standard frame rule with a simplified version of the Independence lemma. The \( \ast \) connective in the frame rule requires that its arguments be disjoint. The Independence lemma generalizes \( \ast \) to least upper bound.

### 5.1 Supporting Lemmas

Figure 4 shows a frame rule, due to O’Hearn et al. [23], which captures the idea that, given a program \( e \) with a precondition \( p \) that holds before it runs and a postcondition \( q \) that holds afterward, a disjoint condition \( r \) that holds before \( e \) runs will continue to hold afterward. Moreover, the original postcondition \( q \) will continue to hold.

For \( \lambda_{\text{LVar}} \), we can state and prove an analogous local reasoning property. Lemma 1, the Independence lemma, says that if the configuration \( \langle S; e \rangle \) steps to \( \langle S'; e' \rangle \), then the configuration \( \langle S \cup S''; e \rangle \), where \( S'' \) is some other store (e.g., one from another subcomputation), can step to \( \langle S'; e' \rangle \).

**Proof sketch.** By induction on the derivation of \( \langle S; e \rangle \rightarrow \langle S'; e' \rangle \), by cases on the last rule in the derivation.

The Clash lemma, Lemma 2, is similar to the Independence lemma, but handles the case where \( S \cup S'' = T_S \). It ensures that in that case, \( \langle S \cup S''; e \rangle \) steps to \( \text{error} \).

**Lemma 2** (Clash). If \( \langle S; e \rangle \rightarrow \langle S'; e' \rangle \) (where \( S'; e' \neq \text{error} \)), then for all \( S'' \) such that \( S'' \text{ non-conflicting with } \langle S; e \rangle \rightarrow \langle S'; e' \rangle \), \( S \cup S'' \subseteq T_S \):

\[
\langle S \cup S''; e \rangle \rightarrow \text{error}.
\]

**Proof sketch.** By induction on the derivation of \( \langle S; e \rangle \rightarrow \langle S'; e' \rangle \), by cases on the last rule in the derivation.

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

**Lemma 3** (Error Preservation). If \( \langle S; e \rangle \rightarrow \text{error} \) and \( S \subseteq S' \), then \( \langle S'; e \rangle \rightarrow \text{error} \).

**Proof sketch.** By induction on the derivation of \( \langle S; e \rangle \rightarrow \text{error} \), by cases on the last rule in the derivation.

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

**Definition 5.** A store \( S'' \) is non-conflicting with the transition \( \langle S; e \rangle \rightarrow \langle S'; e' \rangle \) iff \( \text{dom}(S'') - \text{dom}(S) \cap \text{dom}(S'') = \emptyset \).
By induction hypothesis, there exist \( \sigma_1, \sigma_2 \) such that
\[
\langle S; e_1 \rangle \quad \langle S_a; e_1 \rangle

\begin{align*}
\sigma_1 &= \langle S_a; e_1 \rangle \\
\sigma_2 &= \langle S_b; e_2 \rangle
\end{align*}
\]

To show: There exists \( \sigma_c \) such that
\[
\langle S; e_1 e_2 \rangle \quad \langle S_{a1} \cup_S S_{a2}; e_1 e_2 \rangle

\begin{align*}
\sigma_c &= \langle S_{a1} \cup_S S_{a2}; e_1 e_2 \rangle \\
\sigma &= \langle S; e_1 \rangle
\end{align*}
\]

Figure 6. Diagram of the subcase of Lemma 4 in which the E-PARAPP rule is the last rule in the derivation of both \( \sigma \rightarrow \sigma_a \) and \( \sigma \rightarrow \sigma_b \). We are required to show that, if the configuration \( \langle S; e_1 e_2 \rangle \) steps by E-PARAPP to two different configurations, \( \langle S_{a1} \cup_S S_{a2}; e_1 e_2 \rangle \) and \( \langle S_{b1} \cup_S S_{b2}; e_1 e_2 \rangle \), then they both step to some third configuration \( \sigma_c \).

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

Proof sketch. By induction on \( \sigma \rightarrow \sigma_a \), or cases on the last rule in the derivation. Renaming is only necessary in the E-NEW case.

The most interesting subcase is that in which the E-PARAPP rule is the last rule in the derivation of both \( \sigma \rightarrow \sigma_a \) and \( \sigma \rightarrow \sigma_b \). Here, as Figure 6 illustrates, appealing to the induction hypothesis alone is not enough to complete the case, and Lemmas 1 and 2 both play a role. For instance, suppose we have from the induction hypothesis that \( \langle S_{a1}; e_{a1} \rangle \) steps to \( \langle S_{a1}; e_{a1} \rangle \). To complete the case, we need to show that \( \langle S_{a1} \cup_S S_{a2}; e_{a1} e_{a2} \rangle \) can take a step by the E-PARAPP rule. But for E-PARAPP to apply, we need to show that \( e_{a1} \) can take a step beginning in the larger store of \( S_{a1} \cup_S S_{a2} \). To do so, we can appeal to Lemma 1 which allows us to “frame on” the additional store \( S_{a2} \) that has resulted from a parallel subcomputation.

We can readily restate Lemma 2 as Corollary 1.

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

Proof. Choose \( i = j = 1 \). The proof follows immediately from Lemma 4.

5.3 Confluence Lemmas and Determinism

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

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

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

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

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

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

Proof. Strong Confluence (Lemma 6) implies Confluence.

Theorem 1 (Determinism). If \( \sigma \rightarrow^* \sigma' \) and \( \sigma \rightarrow^* \sigma'' \), and neither \( \sigma' \) nor \( \sigma'' \) can take a step except by E-REFL or E-REFLKER, then \( \sigma' = \sigma'' \).

Proof. We have from Lemma 7 that there exists \( \sigma_c \) such that \( \sigma' \rightarrow^* \sigma_c \) and \( \sigma'' \rightarrow^* \sigma_c \). Since \( \sigma' \) and \( \sigma'' \) can only step to themselves, we must have \( \sigma' = \sigma_c \) and \( \sigma'' = \sigma_c \), hence \( \sigma' = \sigma'' \).

5.4 Discussion: Termination

Above we have followed Budimlić et al.7 in treating determinism separately from the issue of termination. Yet one might legitimately be concerned that in \( \lambda_{\text{Val}} \), a configuration could have both an infinite reduction path and one that terminates with a value. Theorem 1 says that if two runs of a given \( \lambda_{\text{Val}} \) program reach configurations where no more reductions are possible (except by reflexive rules), then they have reached the same configuration. Hence Theorem 1 handles the case of deadlocks already: a \( \lambda_{\text{Val}} \) program can deadlock (e.g., with a blocked get), but it will do so deterministically.

9Lemmas 5 and 7 are nearly identical to the corresponding lemmas in the proof of determinism for Featherweight CnC given by Budimlić et al. 7. We also reuse Budimlić et al.’s naming conventions for Lemmas 1 through 4 but they differ considerably in our setting due to the generality of LVars.
putInSet a limited interface specific to their data structure (cretically, the data structure author imports our library and reexports every run of that program terminates in the LVars library, which provides core runtime functionality: thread equivalent of put data structure form a lattice and that it is only accessible via the data structure author's obligation to ensure that the states of their the application writer receives a determinism guarantee; it is the second, the application writer who uses that data structure. Only Use of our LVars library typically in-

\[ \lambda \]

We have implemented a prototype LVars library based on the \( \lambda \) relationship to benchmarking results, is available in in the LVars repository. The Our library, together with example programs and preliminary error configuration“ condition is necessary because it is possible to construct a given \( \lambda \) program that can terminate in error on some runs and diverge on others. By contrast, our existing determinism theorem does not have to treat error specially.

6. Prototype Implementation and Evaluation

We have implemented a prototype LVars library based on the \( \lambda \) monad-par Haskell library, which provides the \( \lambda \) monad \[ \lambda \]. Our library, together with example programs and preliminary benchmarking results, is available in in the LVars repository. The relationship to \( \lambda \) is somewhat loose; for instance, while evaluation in \( \lambda \) is always strict, our library allows lazy, pure Haskell computations along with strict, parallel monadic computations.

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

The data structure author uses an interface provided by our LVars library, which provides core runtime functionality: thread scheduling and tracking of threads blocked on get operations. Concretely, the data structure author imports our library and reexports a limited interface specific to their data structure (e.g., for sets, putInSet and waitSetSizeThreshold). In fact, our library provides three different runtime interfaces for the data structure author to choose among. These “layered” interfaces provide the data structure author with a shifting trade-off between the ease of meeting their proof obligation, and attainable performance:

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

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

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

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

Revisiting our breath-first traversal example In Section 2 we proposed using LVars to implement a program that performs a breadth-first traversal of a connected component of a graph, mapping a function over each node in the component. Figure 8 gives a version of this program implemented using our LVars library. It performs a breadth-first traversal of the profiles graph with effectful put operations on a shared set variable. This variable, analyzedSet, has to be modified multiple times by putInSet and thus cannot be an IVar. The callback function analyze is “baked into” analyzedSet and may run as soon as new elements are inserted. (In the \( \lambda \) calculus, analyze could be built into the convert function and applied to the result of get.) Our implementation uses the “Pure LVars” runtime layer described above: analyzedSet is nothing more than a tree-based data structure (Data.Set) stored in a mutable location.

Preliminary benchmarking results We compared the performance of the LVar-based implementation of \( \lambda \) traverse against the version in Figure 1 which we ran using the Control.Parallel.Strategies library \[ \lambda \], version 3.2.0.3. (Despite being a simple algorithm, even breadth-first search by itself is considered a useful benchmark; in fact, the well-known “Graph 500” \[ ? \] benchmark is exactly breadth-first search.)

We evaluated the Strategies and LVar versions of \( \lambda \) traverse by running both on a local random directed graph of 40,000 nodes and 320,000 edges (and therefore an average degree of 8), simulating the analyze function by doing a specified amount of work for each node, which we varied from 1 to 32 microseconds. Fig-

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

Figure 8. An example Haskell program, written using our LVars library, that maps a computation over a connected component using a monotonically growing set variable. The code is written in a strict style, using the Par monad for parallelism. The use of the set variable enables modularity and safe pipelining. Consumers can safely asynchronously execute work items put into analyzedSet.
```

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

\[ \lambda \]
Figure 9 shows the results of our evaluation on 1, 2, 3, and 4 cores. Although both the Strategies and LVar versions enjoyed a speedup as we added parallel resources, the LVar version scaled particularly well. A subtler, but interesting point is that, in the Strategies version, it took an average of 64.64 milliseconds for the first invocation of analyze to begin running after the program began, whereas in the LVar version, it took an average of only 0.18 milliseconds, indicating that the LVar version allows work to be pipelined.

7. Safe, Limited Nondeterminism

In practice, a major problem with nondeterministic programs is that they can silently go wrong. Most parallel programming models are unsafe in this sense, but we may classify a nondeterministic language as safe if all occurrences of nondeterminism—that is, execution paths that would yield a wrong answer—are trapped and reported as errors. This notion of safe nondeterminism is analogous to the concept of type safety: type-safe programs can throw exceptions, but they will not “go wrong”. We find that there are various extensions to a deterministic language that make it safely nondeterministic\(^{12}\). Here, we will look at one such extension: exact but destructive observations.

We begin by noting that when the state of an LVar has come to rest—when no more puts will occur—then its final value is a deterministic function of program inputs, and is therefore safe to read directly, rather than through a thresholded get. For instance, once no more elements will be added to the \(\_\_\_\_\_\_\_\_\text{acc}\) accumulator variable in the \texttt{bf\_traverse} example of Figure\(^8\) it is safe to read the exact, complete set contents.

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

\[
\text{probation} \leq T \quad \forall d \in D. \; d \nleq \text{probation}
\]

We then propose a new operation, \texttt{consume}, that takes a pointer to an LVar \(l\), updates the store, setting \(l\)’s state to \texttt{probation}, and returns a singleton set containing the \texttt{exact} previous state of \(l\), rather than a lower bound on that state. The idea is to ensure that, after a \texttt{consume}, any further operations on \(l\) will go awry; put operations will attempt to move the state of \(l\) to \(T\), resulting in error.

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

\begin{verbatim}
let \(\texttt{cnt} = \text{new in}
\texttt{let \sum = \text{new in}
\texttt{let \par \texttt{p1 = (bump3 sum; put \{a\} cnt)}
\texttt{p2 = (bump2 sum; put \{b\} cnt)}
\texttt{p3 = (bump1 sum; put \{c\} cnt)}
\texttt{r = (get cnt \{a, b, c\}; consume \sum)}
\texttt{in ... r ...}
\end{verbatim}

(Example 3)

In [Example 3], we use semicolon for sequential composition: \(e_1; e_2\) is sugar for \texttt{let _ = \(e_1\) in \(e_2\)}. We also assume a new syntactic sugar in the form of a \texttt{bump} operation that takes a pointer to an LVar representing a counter and increments it by one, with \texttt{bump} as an additional shorthand for \(n\) consecutive \texttt{bumps} to \(l\). Meanwhile, the \texttt{cnt} LVar uses the power-set lattice of the set \(\{a, b, c\}\) to track the completion of the \(p_1, p_2\) and \(p_3\) “threads”.

\(^{12}\) For instance, while not recognized explicitly by the authors as such, a recent extension to CnC for memory management\(^{25}\) incidentally fell into this category.
Before the call to consume, get cnt \{a, b, c\} serves as a synchroniza-
tion mechanism, ensuring that all increments are complete before the value is read. Three writers and one reader execute in parallel, and only when all writers complete does the reader return the sum, which in this case will be 3 + 4 + 5 = 12.

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

7.1 Desugaring bump

In this section we explain how the bump operation for LVar counters—as well as an underlying capacity for generating unique IDs—can be implemented in \(\lambda_{\text{LVar}}\).

Strictly speaking, if we directly used the atomic counter lattice of Figure 2(c) for the sum LVar in Example 3, we would not be able to implement bump. Rather than use that lattice directly, then, we can simulate it using a power-set lattice over an arbitrary alphabet of symbols \(\{s_1, s_2, s_3, \ldots\}\), ordered by subset inclusion. LVars whose states occupy such a lattice encode natural numbers using the cardinality of the subset. With this encoding, incrementing a unique \(\lambda\)’s semantics would involve parameterizing the type system by its unique cardinality. Here we use \(\{s, s, s\}\), \(\{s, s, s, s\}\), \(\{s, s, s, s, s\}\), and \(\{s, s, s, s, s, s\}\) ordered by subset inclusion to form a unique \(\lambda\) number with \(\lambda\) atoms. The expression \(\text{sum} l\) could then simply desugary to put \(l\).

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

Figure 10 gives one possible set of rewrite rules for a transformation that uses the pedigree technique to desugar \(\lambda_{\text{LVar}}\) programs containing unique. Bearing some resemblance to a continuation-passing-style transformation \[11\], it creates a tree that tracks the dynamic evaluation of applications. We have implemented this transformation as a part of our interpreter for \(\lambda_{\text{LVar}}\) extended with consume. With unique in place, we can write programs like the following, in which two parallel computations increment the same counter:

\[\begin{align*}
\text{let } p_1 &= (\text{put sum unique; put sum unique}) \\
\text{p}_2 &= (\text{put sum unique}) \\
\text{in }...
\end{align*}\]

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

8. Related Work

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

Deterministic Parallel Java (DPJ) DPJ \[6\] is a deterministic language consisting of a system of annotations for Java code. A sophisticated region-based type system ensures that a mutable region of the heap is, essentially, passed linearly to an exclusive writer. While a linear type system or region system like that of DPJ could be used to enforce single assignment statically, accommodating \(\lambda_{\text{LVar}}\)’s semantics would involve parameterizing the type system by the user-specified lattice.

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

Concurrent Revisions The Concurrent Revisions (CR) \[19\] modeling program uses isolation types to distinguish regions of the heap shared by multiple mutators. Rather than enforcing exclusive access, CR clones a copy of the state for each mutator, using a deterministic policy for resolving conflicts in local copies. The management of shared variables in CR is tightly coupled to a fork/join control structure, and the implementation of these variables is similar to reduction variables in other languages (e.g., Cilk hyperobjects). CR charts an important new area in the deterministic-parallelism design space, but one that differs significantly from...
References


[2] The Parallel Boost Graph Library. URL http://www.boost.org/doc/libs/1_63_0/libs/graph_parallel


A. Safe Renaming

When $\lambda_{\text{Var}}$ programs split into two subcomputations via the E-PARAPP rule, the subcomputations’ stores are merged (via the lub operation) as they are running. Therefore we need to ensure that the following two properties hold:

1. Location names created before a split still match up with each other after a merge.
2. Location names created by each subcomputation while they are running independently do not match up with each other accidentally—i.e., they do not collide.

Property (2) is why it is necessary to rename locations in the E-PARAPP (and E-PARAPPERR) rule. This renaming is accomplished by a call to the rename metafunction, which, for each location name $l$ generated during the reduction $\langle S; e_1 \rangle \leadsto \langle S_1; e_1' \rangle$, generates a name that is not yet used on either side of the split and substitutes that name into $\langle S_1; e_1' \rangle$. We arbitrarily choose to rename locations created during the reduction of $\langle S_1; e_1 \rangle$, but it would work just as well to rename those created during the reduction of $\langle S_1; e_2 \rangle$.

**Definition 6.** The rename metafunction is defined as follows:

$$\text{rename}(\cdot, \cdot, \cdot) : \sigma \times S \times S \rightarrow \sigma$$

$$\text{rename}(\langle S'; e \rangle, \langle S'' \rangle, S) \triangleq \langle S'' \rangle[1 \mapsto l_1'] \ldots [n \mapsto l_n']$$

where:

- $\{l_1, \ldots, l_n\} = \text{dom}(S') - \text{dom}(S)$, and
- $\{l_1', \ldots, l_n'\}$ is a set such that $l_i' \notin (\text{dom}(S') \cup \text{dom}(S''))$ for $i \in [1..n]$.

However, property (1) means that we cannot allow $\alpha$-renaming of bound locations in a configuration to be done at will. Rather, renaming can only be done safely if it is done in the context of a transition from configuration to configuration. Therefore, we define a notion of safe renaming with respect to a transition.

**Definition 7.** A renaming of a configuration $\langle S; e \rangle$ is the substitution into $\langle S; e \rangle$ of location names $l_1', \ldots, l_n'$ for some subset $l_1, \ldots, l_n$ of $\text{dom}(S)$.

**Definition 8.** A safe renaming of $\langle S'; e' \rangle$ with respect to $\langle S; e \rangle \leadsto \langle S'; e' \rangle$ is a renaming of $\langle S'; e' \rangle$ in which the locations $l_1, \ldots, l_n$ being renamed are the members of the set $\text{dom}(S') - \text{dom}(S)$, and the names $l_1', \ldots, l_n'$ that are replacing $l_1, \ldots, l_n$ do not appear in $\text{dom}(S')$.

If $(S''; e'')$ is a safe renaming of $(S'; e')$ with respect to $(S; e) \leadsto (S'; e')$, then $S''$ is by definition non-conflicting with $(S; e) \leadsto (S'; e')$.

### A.1 Renaming Lemmas

With the aforementioned definitions in place, we can establish the following two properties about renaming. **Lemma 8** expresses the idea that the names of locations created during a reduction step are arbitrary within the context of that step. It says that if a configuration $\langle S; e \rangle$ steps to $\langle S'; e' \rangle$, then $\langle S; e \rangle$ can also step to configurations that are safe renamings of $\langle S'; e' \rangle$ with respect to $\langle S; e \rangle$.

**Lemma 8 (Renaming of Locations During a Step).** If $(S; e) \leadsto (S'; e')$ (where $(S'; e') \neq \text{error}$) and $\{l_1, \ldots, l_n\} = \text{dom}(S') - \text{dom}(S)$, then:

For all sets $\{l_1', \ldots, l_n'\}$ such that $l_i' \notin \text{dom}(S')$ for $i \in [1..n]$:

$$(S; e) \leadsto (S_{\text{oldloc}}[l_1 \mapsto S'(l_1)] \ldots [l_n \mapsto S'(l_n)]; e'[l_1 \mapsto l_1'] \ldots [l_n \mapsto l_n'])$$

$$(\neq \text{error}),$$

where $S_{\text{oldloc}}$ is defined as follows: $\text{dom}(S_{\text{oldloc}}) = \text{dom}(S)$, and for all $l \in \text{dom}(S_{\text{oldloc}})$, $S_{\text{oldloc}}(l) = S'(l)$.

**Proof sketch.** By induction on the derivation of $(S; e) \leadsto (S'; e')$, by cases on the last rule in the derivation.

Finally, **Lemma 9** says that in the circumstances where we use the rename metafunction, the renaming it performs meets the specification set by **Lemma 8**.

**Lemma 9 (Safety of rename).** If $(S; e) \leadsto (S'; e')$ (where $(S'; e') \neq \text{error}$ and $S'' \neq \top$), then:

$$(S; e) \leadsto \text{rename}(\langle S'; e' \rangle, S'').$$

**Proof sketch.** Straightforward application of **Lemma 8**

---

14 Since $\lambda_{\text{Var}}$ locations are drawn from a distinguished set $\text{Loc}$, they cannot occur in the user-specified $D$—that is, locations in $\lambda_{\text{Var}}$ may not contain pointers to other locations. Likewise, $\lambda$-bound variables in $e$ are never location names. Therefore, substitutions like the one in **Definition 6** will not capture bound occurrences of location names.