LVars: Lattice-based Data Structures for Deterministic Parallelism

Lindsey Kuper Ryan R. Newton
Indiana University
{lkuper, rrnewton}@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 `pseq` combinators can provide real speedups on practical programs while guaranteeing determinism [21].[1] Yet pure programming with futures is not ideal for all problems. Consider a *producer/consumer* computation in which producers and consumers can be scheduled onto separate processors, each able to keep their working sets in cache. Such a scenario enables *pipeline parallelism* and is common, for instance, in stream processing. But a clear separation of producers and consumers is difficult with futures, because whenever a consumer forces a future, if it is not yet available, the consumer immediately switches roles to begin computing the value (as explored in previous work [22]).

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.[2] 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 monotonic 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 [18], 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 *IVars* [4] and are a well-established

FHPC '13, September 23, 2013, Boston, MA, USA.

ACM 978-1-4503-2381-9/13/09…$15.00.
http://dx.doi.org/10.1145/2502323.2502326

[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 Qthreads 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 `IO`, so that such communication can occur anywhere inside pure programs.[3]

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 2.

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*.[4] 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 $\top$ 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 λ_{LVar}, a deterministic parallel calculus with shared state, based on the call-by-value λ-calculus. The λ_{LVar} language is general enough to subsume existing deterministic parallel languages because it is parameterized by the choice of lattice. For example, a lattice of channel histories with a prefix ordering allows LVars to represent FIFO channels that implement a Kahn process network, whereas instantiating λ_{LVar} with a lattice with "empty" and "full" states (where $empty < full$) results in a parallel 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 λ_{LVar} determinism result.

Contributions

- We introduce LVars (Section 3) and use them to define λ_{LVar}, a parallel calculus that uses LVars for shared state (Section 4). We have implemented λ_{LVar} as a runnable PLT Redex model.
- As our main technical result, we give a proof of determinism for λ_{LVar} (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 λ_{LVar} 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 λ_{LVar} 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 λ_{LVar} 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 6).

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

[3] That is, monad-par provides a `Par` monad, exposing effectful `put` and `get` operations on IVars, but with a `runPar` method similar to `runST`.

[4] As we will see in Section 3.1, this "lattice" need only be a *bounded join-semilattice* augmented with a greatest element $\top$, in which every two elements have a least upper bound but not necessarily a greatest lower bound. For brevity, we use the term "lattice" here and in the rest of this paper.

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

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 `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 `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.[5]

A compromise would be for `bf_traverse` to return a list of level-sets: distance one nodes, distance two nodes, and so on. Thus level-one results could be consumed before level-two results are ready. Still, the problem would remain: within each level-set, we cannot launch all instances of `analyze` and asynchronously use those results that 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 λ_{LVar} calculus and giving its determinism proof in the next few

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

```
nbrs :: Graph → NodeLabel → 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.
bf_traverse :: Graph → Set NodeLabel →
               Set NodeLabel → Set NodeLabel
bf_traverse g seen nu =
  if nu == {}
  then seen
  else let seen'  = union seen nu
           allNbr = parFold union (parMap (nbrs g) nu)
           nu'    = difference allNbr seen'
       in bf_traverse g seen' nu'

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

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

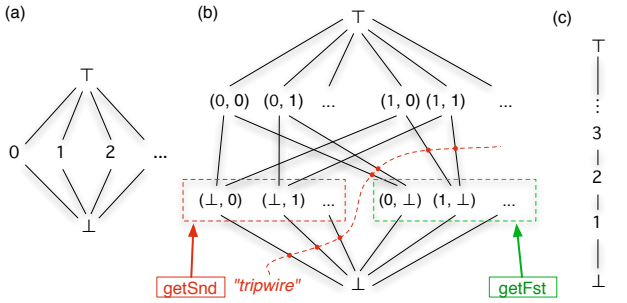

Figure 2. Example lattices: (a) IVar containing a natural number; (b) pair of natural-number-valued IVars; (c) positive integers ordered by $\leq$ (see Section 7.1). Subfigure (b) is annotated with example threshold sets that would correspond to a blocking read of the first or second element of the pair (see Sections 3.3 and 4.2). Any state transition crossing the "tripwire" for `getSnd` causes it to unblock and return a result.

sections, in Section 6 we give an LVar-based solution to our challenge problem, implemented using our Haskell LVars library, along with a performance evaluation.

3. Lattices, Stores, and Determinism

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

In this setting of shared mutable state, the trick that λ_{LVar} employs to maintain determinism is that stores contain *LVars*, which are a generalization of IVars.[6] Whereas IVars are single-assignment

[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 $\sqsubseteq$. An LVar can take on any sequence of states from D, so long as that sequence respects the partial order—that is, updates to the LVar (made via the `put` operation) are *inflationary* with respect to $\sqsubseteq$. Moreover, the `get` operation allows only limited observations of the LVar's state. In this section, we discuss how lattices and stores work in λ_{LVar} and explain how the semantics of `put` and `get` together enforce determinism in λ_{LVar} programs.

3.1 Lattices

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

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

- D has a least element $\bot$, representing the initial "empty" state of an LVar.
- D has a greatest element $\top$, representing the "error" state that results from conflicting updates to an LVar.
- D comes equipped with a partial order $\sqsubseteq$, where $\bot \sqsubseteq d \sqsubseteq \top$ for all $d \in D$.
- Every pair of elements in D has a least upper bound (lub) $\sqcup$. Intuitively, the existence of a lub for every two elements in D means that it is possible for two subcomputations to independently update an LVar, and then deterministically merge the results by taking the lub of the resulting two states.

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

Figure 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,

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

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

In the lattice of Figure 2(c), on the other hand, the $\top$ state can never be reached, because the least upper bound of any two writes is just the maximum of the two. For instance, if one thread writes 2 and another writes 1, the resulting state will be 2, since $2 \sqcup 1 = 2$. Here, the unreachability of $\top$ models the fact that no conflicting updates can occur to the LVar.

3.2 Stores

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

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

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

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

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

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

- $S_1 \sqcup_S S_2 = \top_S$ iff there exists some $l \in dom(S_1) \cap dom(S_2)$ such that $S_1(l) \sqcup S_2(l) = \top$.
- 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 $l \in dom(S)$:

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

By Definition 3, if $d_1 \sqcup d_2 = \top$, then $[l \mapsto d_1] \sqcup_S [l \mapsto d_2] = \top_S$. Notice that a store containing a binding $l \mapsto \top$ can never arise during the execution of a λ_{LVar} program, because (as we will see in Section 4) 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 $\bot$, and returns the location l of that LVar (*i.e.*, a pointer to the LVar).
- `put` takes a pointer to an LVar and a singleton set containing a new state and updates the LVar's state to the *least upper bound* of the current state and the new state, potentially pushing the state of the LVar upward in the lattice. Any update that would take the state of an LVar to $\top$ results in an error.
- `get` performs a blocking "threshold" read that allows limited observations of the state of an LVar. It takes a pointer to an LVar and a *threshold set* Q, which is a non-empty subset of D that is *pairwise incompatible*, meaning that the lub of any two distinct elements in Q is $\top$. If the LVar's state d in the lattice is *at or above* some $d' \in Q$, the `get` operation unblocks and returns the singleton set $\{d'\}$. Note that d' is a unique element of Q, for if there is another $d'' \neq d'$ in the threshold set such that $d'' \sqsubseteq d$, it would follow that $d' \sqcup d'' \sqsubseteq d \neq \top$, which contradicts the requirement that Q be pairwise incompatible.

The intuition behind `get` is that it specifies a subset of the lattice that is "horizontal": no two elements in the threshold set can be above or below one another. Intuitively, each element in the threshold set is an "alarm" that detects the activation of itself or any state above it. One way of visualizing the threshold set for a `get` operation is as a subset of edges in the lattice that, if crossed, set off the corresponding alarm. Together these edges form a "tripwire". This visualization is pictured in Figure 2(b). The threshold set $\{(\bot, 0), (\bot, 1), ...\}$ (or a subset thereof) would pass the incompatibility test, as would the threshold set $\{(0, \bot), (1, \bot), ...\}$ (or a subset thereof), but a combination of the two would not pass.

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

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:[7]

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

Our Definition 2 is reminiscent of Definition 4, but Definition 4 requires that $S(l)$ and $S'(l)$ be *equal*, instead of our weaker requirement that $S(l) \sqsubseteq S'(l)$ according to the user-provided partial order $\sqsubseteq$. In λ_{LVar}, stores may grow by updating existing bindings via repeated `puts`, so Definition 4 would be too strong; for instance, if $\bot \sqsubset d_1 \sqsubseteq d_2$ for distinct $d_1, d_2 \in D$, the relationship $[l \mapsto d_1] \sqsubseteq_S [l \mapsto d_2]$ holds under Definition 2, but would not hold under Definition 4. That is, in λ_{LVar} an LVar could take on the state d_1 followed by d_2, which would not be possible in Featherweight CnC. We establish in Section 5 that λ_{LVar} remains deterministic despite the relatively weak $\sqsubseteq_S$ 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. λ_{LVar}: Syntax and Semantics

The syntax and operational semantics of λ_{LVar} appear in Figures 3 and 4, respectively.[8] As we have noted, both the syntax and semantics are parameterized by the lattice D. The reduction relation $\longrightarrow$ is defined on *configurations* $\langle S;\ e\rangle$ comprising a store and an expression. The *error configuration*, written **error**, is a unique element added to the set of configurations, but we consider $\langle \top_S;\ e\rangle$ to be equal to **error**, for all expressions e. The metavariable σ ranges over configurations.

Figure 4 shows two disjoint sets of reduction rules: those that step to configurations other than **error**, and those that step to **error**. Most of the latter set of rules merely propagate errors along. A new **error** can only arise by way of the E-PARAPPERR rule, which represents the joining of two conflicting subcomputations, or by way of the E-PUTVALERR rule, which applies when a `put` to a location would take its state to $\top$.

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

[8] We have implemented λ_{LVar} as a runnable PLT Redex [12] model, available in the LVars repository.

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

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

Figure 3. Syntax for λ_{LVar}.

The rules E-NEW, E-PUTVAL/E-PUTVALERR, and E-GETVAL respectively express the semantics of the `new`, `put`, and `get` operations described in Section 3.3. The incompatibility property of the threshold set argument to `get` is enforced in the E-GETVAL rule by the $incomp(Q)$ 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

λ_{LVar} has an explicitly parallel reduction semantics: the E-PARAPP rule in Figure 4 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, λ_{LVar} is still call-by-value in the sense that arguments must be fully evaluated before function application (β-reduction, modeled by the E-BETA rule) can occur. We can exploit this property to define a syntactic sugar `let par` for *parallel composition*, which computes two subexpressions e_1 and e_2 in parallel before computing e_3:

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

Although e_1 and e_2 can be evaluated in parallel, e_3 cannot be evaluated until both e_1 and e_2 are values, because the call-by-value semantics does not allow β-reduction until the operand is fully evaluated, and because it further disallows reduction under λ-terms (sometimes called "full β-reduction"). In the terminology

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

$incomp(Q) \triangleq \forall\, a, b \in Q.\ (a \neq b \implies a \sqcup b = \top)$

$\boxed{\langle S;\, e\rangle \longrightarrow \langle S';\, e'\rangle}$ (where $\langle S';\, e'\rangle \neq$ **error**)

$$\text{E-Refl}\ \frac{}{\langle S;\, e\rangle \longrightarrow \langle S;\, e\rangle} \qquad \text{E-ParApp}\ \frac{\langle S;\, e_1\rangle \longrightarrow \langle S_1;\, e_1'\rangle \quad \langle S;\, e_2\rangle \longrightarrow \langle S_2;\, e_2'\rangle \quad \langle S_1^r;\, e_1'^r\rangle = rename(\langle S_1;\, e_1'\rangle, S_2, S) \quad S_1^r \sqcup_S S_2 \neq \top_S}{\langle S;\, e_1\ e_2\rangle \longrightarrow \langle S_1^r \sqcup_S S_2;\, e_1'^r\ e_2'\rangle}$$

$$\text{E-Put-1}\ \frac{\langle S;\, e_1\rangle \longrightarrow \langle S_1;\, e_1'\rangle}{\langle S;\, \texttt{put}\ e_1\ e_2\rangle \longrightarrow \langle S_1;\, \texttt{put}\ e_1'\ e_2\rangle} \qquad \text{E-Put-2}\ \frac{\langle S;\, e_2\rangle \longrightarrow \langle S_2;\, e_2'\rangle}{\langle S;\, \texttt{put}\ e_1\ e_2\rangle \longrightarrow \langle S_2;\, \texttt{put}\ e_1\ e_2'\rangle}$$

$$\text{E-PutVal}\ \frac{S(l) = d_2 \quad d_1 \in D \quad d_1 \sqcup d_2 \neq \top}{\langle S;\, \texttt{put}\ l\ \{d_1\}\rangle \longrightarrow \langle S[l \mapsto d_1 \sqcup d_2];\, \{\}\rangle}$$

$$\text{E-Get-1}\ \frac{\langle S;\, e_1\rangle \longrightarrow \langle S_1;\, e_1'\rangle}{\langle S;\, \texttt{get}\ e_1\ e_2\rangle \longrightarrow \langle S_1;\, \texttt{get}\ e_1'\ e_2\rangle} \qquad \text{E-Get-2}\ \frac{\langle S;\, e_2\rangle \longrightarrow \langle S_2;\, e_2'\rangle}{\langle S;\, \texttt{get}\ e_1\ e_2\rangle \longrightarrow \langle S_2;\, \texttt{get}\ e_1\ e_2'\rangle}$$

$$\text{E-GetVal}\ \frac{S(l) = d_2 \quad incomp(Q) \quad Q \subseteq D \quad d_1 \in Q \quad d_1 \sqsubseteq d_2}{\langle S;\, \texttt{get}\ l\ Q\rangle \longrightarrow \langle S;\, \{d_1\}\rangle}$$

$$\text{E-Convert}\ \frac{\langle S;\, e\rangle \longrightarrow \langle S';\, e'\rangle}{\langle S;\, \texttt{convert}\ e\rangle \longrightarrow \langle S';\, \texttt{convert}\ e'\rangle} \qquad \text{E-ConvertVal}\ \frac{}{\langle S;\, \texttt{convert}\ v\rangle \longrightarrow \langle S;\, \delta(v)\rangle} \qquad \text{E-Beta}\ \frac{}{\langle S;\, (\lambda x.\, e)\ v\rangle \longrightarrow \langle S;\, e[x := v]\rangle}$$

$$\text{E-New}\ \frac{}{\langle S;\, \texttt{new}\rangle \longrightarrow \langle S[l \mapsto \bot];\, l\rangle}\ (l \notin dom(S))$$

$\boxed{\langle S;\, e\rangle \longrightarrow \textbf{error}}$

$$\text{E-ReflErr}\ \frac{}{\textbf{error} \longrightarrow \textbf{error}} \qquad \text{E-ParAppErr}\ \frac{\langle S;\, e_1\rangle \longrightarrow \langle S_1;\, e_1'\rangle \quad \langle S;\, e_2\rangle \longrightarrow \langle S_2;\, e_2'\rangle \quad \langle S_1^r;\, e_1'^r\rangle = rename(\langle S_1;\, e_1'\rangle, S_2, S) \quad S_1^r \sqcup_S S_2 = \top_S}{\langle S;\, e_1\ e_2\rangle \longrightarrow \textbf{error}}$$

$$\text{E-AppErr-1}\ \frac{\langle S;\, e_1\rangle \longrightarrow \textbf{error}}{\langle S;\, e_1\ e_2\rangle \longrightarrow \textbf{error}} \qquad \text{E-AppErr-2}\ \frac{\langle S;\, e_2\rangle \longrightarrow \textbf{error}}{\langle S;\, e_1\ e_2\rangle \longrightarrow \textbf{error}}$$

$$\text{E-PutErr-1}\ \frac{\langle S;\, e_1\rangle \longrightarrow \textbf{error}}{\langle S;\, \texttt{put}\ e_1\ e_2\rangle \longrightarrow \textbf{error}} \qquad \text{E-PutErr-2}\ \frac{\langle S;\, e_2\rangle \longrightarrow \textbf{error}}{\langle S;\, \texttt{put}\ e_1\ e_2\rangle \longrightarrow \textbf{error}}$$

$$\text{E-PutValErr}\ \frac{S(l) = d_2 \quad d_1 \in D \quad d_1 \sqcup d_2 = \top}{\langle S;\, \texttt{put}\ l\ \{d_1\}\rangle \longrightarrow \textbf{error}} \qquad \text{E-GetErr-1}\ \frac{\langle S;\, e_1\rangle \longrightarrow \textbf{error}}{\langle S;\, \texttt{get}\ e_1\ e_2\rangle \longrightarrow \textbf{error}}$$

$$\text{E-GetErr-2}\ \frac{\langle S;\, e_2\rangle \longrightarrow \textbf{error}}{\langle S;\, \texttt{get}\ e_1\ e_2\rangle \longrightarrow \textbf{error}} \qquad \text{E-ConvertErr}\ \frac{\langle S;\, e\rangle \longrightarrow \textbf{error}}{\langle S;\, \texttt{convert}\ e\rangle \longrightarrow \textbf{error}}$$

Figure 4. An operational semantics for λ_{LVar}.

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

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

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

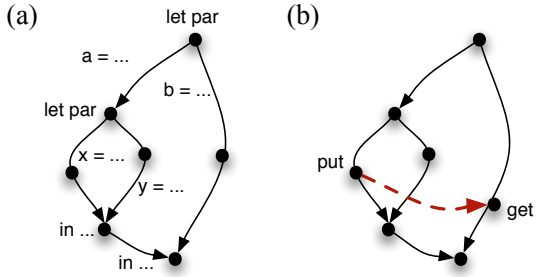

Figure 5. A series-parallel graph induced by parallel λ-calculus evaluation (a); a non-series-parallel graph induced by `put`/`get` operations (b).

4.2 Programming with `put` and `get`

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

```
let p = new in
  let _ = put p {(3,4)} in
    let v1 = get p {(⊥,n) | n ∈ ℕ} in
      ... v1 ...
```
(Example 1)

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

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

our lattice of pairs:

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

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

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

Blocking reads On the other hand, consider the following:

```
let p = new in
  let _ = put p {(⊥, 4)} in
    let par v1 = getFst p                    (Example 2)
            _  = put p {(3, 4)}
      in ... v1 ...
```

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 λ-terms and Back

There are two worlds that λ_{LVar} values may inhabit: the world of threshold sets, and the world of λ-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 λ_{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*, $\delta : v \to v$, exposed 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 λ_{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 λ_{LVar}

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

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

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

Lemma 1 (Independence), simplified:

$$\frac{\langle S;\ e\rangle \longrightarrow \langle S';\ e'\rangle}{\langle S \sqcup_S S'';\ e\rangle \longrightarrow \langle S' \sqcup_S S'';\ e'\rangle}\ (S'' \text{ non-conflicting with } \langle S;\ e\rangle \longrightarrow \langle S';\ e'\rangle)$$

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

5.1 Supporting Lemmas

Figure 7 shows a *frame rule*, due to O'Hearn *et al.* [23], which captures the idea that, given a program c with a precondition p that holds before it runs and a postcondition q that holds afterward, a disjoint condition r that holds before c runs will continue to hold afterward. Moreover, the original postcondition q will continue to hold. For λ_{LVar}, we can state and prove an analogous *local reasoning* property. Lemma 1, the Independence lemma, says that if the configuration $\langle S;\ e\rangle$ can step to $\langle S';\ e'\rangle$, then the configuration $\langle S \sqcup_S S'';\ e\rangle$, where S'' is some other store (*e.g.*, one from another subcomputation), can step to $\langle S' \sqcup_S S'';\ e'\rangle$. Roughly speaking, the Independence lemma allows us to "frame on" a larger store around S and still finish the transition with the original result e', which is key to being able to carry out the determinism proof.

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

Proof sketch. By induction on the derivation of $\langle S;\ e\rangle \longrightarrow \langle S';\ e'\rangle$, by cases on the last rule in the derivation. □

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

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

Proof sketch. By induction on the derivation of $\langle S;\ e\rangle \longrightarrow \langle S';\ e'\rangle$, by cases on the last rule in the derivation. □

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

Lemma 3 (Error Preservation)**.** *If* $\langle S;\ e\rangle \longrightarrow$ **error** *and* $S \sqsubseteq_S S'$*, then* $\langle S';\ e\rangle \longrightarrow$ **error**.

Proof sketch. By induction on the derivation of $\langle S;\ e\rangle \longrightarrow$ **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 $\langle S;\ e\rangle \longrightarrow \langle S';\ e'\rangle$. We say that a store S'' is *non-conflicting* with a transition $\langle S;\ e\rangle \longrightarrow \langle S';\ e'\rangle$ iff $dom(S'')$ does not have any elements in common with $dom(S') - dom(S)$, which is the set of names of *new* store bindings created between $\langle S;\ e\rangle$ and $\langle S';\ e'\rangle$.

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

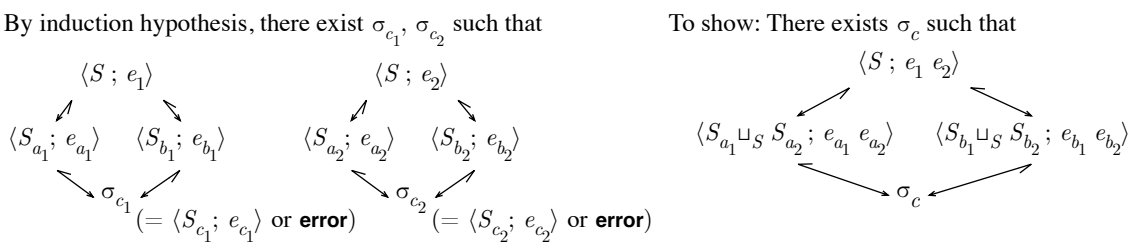

Figure 6. Diagram of the subcase of Lemma 4 in which the E-PARAPP rule is the last rule in the derivation of both $\sigma \longrightarrow \sigma_a$ and $\sigma \longrightarrow \sigma_b$. We are required to show that, if the configuration $\langle S;\ e_1\ e_2\rangle$ steps by E-PARAPP to two different configurations, $\langle S_{a_1} \sqcup_S S_{a_2};\ e_{a_1}\ e_{a_2}\rangle$ and $\langle S_{b_1} \sqcup_S S_{b_2};\ e_{b_1}\ e_{b_2}\rangle$, then they both step to some third configuration σ_c.

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

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

5.2 Diamond Lemma

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

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

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

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

The most interesting subcase is that in which the E-PARAPP rule is the last rule in the derivation of both $\sigma \longrightarrow \sigma_a$ and $\sigma \longrightarrow \sigma_b$. Here, as Figure 6 illustrates, appealing to the induction hypothesis alone is not enough to complete the case, and Lemmas 1, 2, and 3 all play a role. For instance, suppose we have from the induction hypothesis that $\langle S_{a_1};\ e_{a_1}\rangle$ steps to $\langle S_{c_1};\ e_{c_1}\rangle$. To complete the case, we need to show that $\langle S_{a_1} \sqcup_S S_{a_2};\ e_{a_1}\ e_{a_2}\rangle$ can take a step by the E-PARAPP rule. But for E-PARAPP to apply, we need to show that e_{a_1} can take a step beginning in the *larger* store of $S_{a_1} \sqcup_S S_{a_2}$. To do so, we can appeal to Lemma 1, which allows us to "frame on" the additional store S_{a_2} that has resulted from a parallel subcomputation. □

We can readily restate Lemma 4 as Corollary 1:

Corollary 1 (Strong Local Confluence). *If $\sigma \longrightarrow \sigma'$ and $\sigma \longrightarrow \sigma''$, then there exist σ_c, i, j such that $\sigma' \longrightarrow^i \sigma_c$ and $\sigma'' \longrightarrow^j \sigma_c$ 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. [9]

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

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

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

Proof. We have from Lemma 7 that there exists σ_c such that $\sigma' \longrightarrow^* \sigma_c$ and $\sigma'' \longrightarrow^* \sigma_c$. Since σ' and σ'' 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 λ_{LVar}, a configuration could have both an infinite reduction path and one that terminates with a value. Theorem 1 says that if two runs of a given λ_{LVar} program reach configurations where no more reductions are possible (except by reflexive rules), then they have reached the same configuration. Hence Theorem 1 handles the case of *deadlocks* already: a λ_{LVar} program can deadlock (*e.g.*, with a blocked `get`), but it will do so deterministically.

[9] Lemmas 5, 6, and 7 are nearly identical to the corresponding lemmas in the proof of determinism for Featherweight CnC given by Budimlić *et al.* [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.

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

Figure 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 λ_{LVar} program terminates with a non-**error** result, then every run will. We conjecture (but do not prove) that such a consistent termination property holds for λ_{LVar}. Such a property could be paired with Theorem 1 to guarantee that if one run of a given λ_{LVar} program terminates in a non-**error** configuration σ, then every run of that program terminates in σ. (The "non-**error** configuration" condition is necessary because it is possible to construct a λ_{LVar} program that can terminate in **error** on some runs and diverge on others. By contrast, our existing determinism theorem does not have to treat **error** specially.)

6. Prototype Implementation and Evaluation

We have implemented a prototype LVars library based on the *monad-par* Haskell library, which provides the `Par` monad [22]. Our library, together with example programs and preliminary benchmarking results, is available in in the LVars repository. The relationship to λ_{LVar} is somewhat loose: for instance, while evaluation in λ_{LVar} is always strict, our library allows lazy, pure Haskell computations along with strict, parallel monadic computations.

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

The data structure author uses an interface provided by our LVars library, which provides core runtime functionality: thread scheduling and tracking of threads blocked on `get` operations. Concretely, the data structure author imports our library and reexports a limited interface specific to their data structure (*e.g.*, for sets, `putInSet` and `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`[10] function for it and a threshold predicate for each `get` operation. These pure functions are easiest to validate, for example, using the popular QuickCheck [**?**] tool.
2. **IO LVars**: Pure data structures in mutable containers cannot always provide the best performance for concurrent data structures. Thus we provide a more effectful interface. With it, data structures are represented in terms of arbitrary mutable state; performing a `put` requires an *update action* (`IO`) and a `get` requires an effectful polling function that will be run after any `put` to the LVar, to determine if the `get` can unblock.
3. **Scalable LVars**: Polling each blocked `get` upon *any* `put` is not very precise. If the data structure author takes on yet more responsibility, they can use our third interface to reduce contention by managing the storage of blocked computations and threshold functions *themselves*. For example, a concurrent set might store a waitlist of blocked continuations on a per-element basis, rather than using one waitlist for the entire LVar, as in layers (1) and (2).

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

Revisiting our breadth-first traversal example In Section 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.[11] The callback function `analyze` is "baked into" `analyzedSet` and may run as soon as new elements are inserted. (In the λ_{LVar} 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 `bf_traverse` against the version in Figure 1, which we ran using the `Control.Parallel.Strategies` library [21], version 3.2.0.3. (Despite being a simple algorithm, even breadth-first search by itself is considered a useful benchmark; in fact, the well-known "Graph 500" [**?**] benchmark is exactly breadth-first search.)

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

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

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

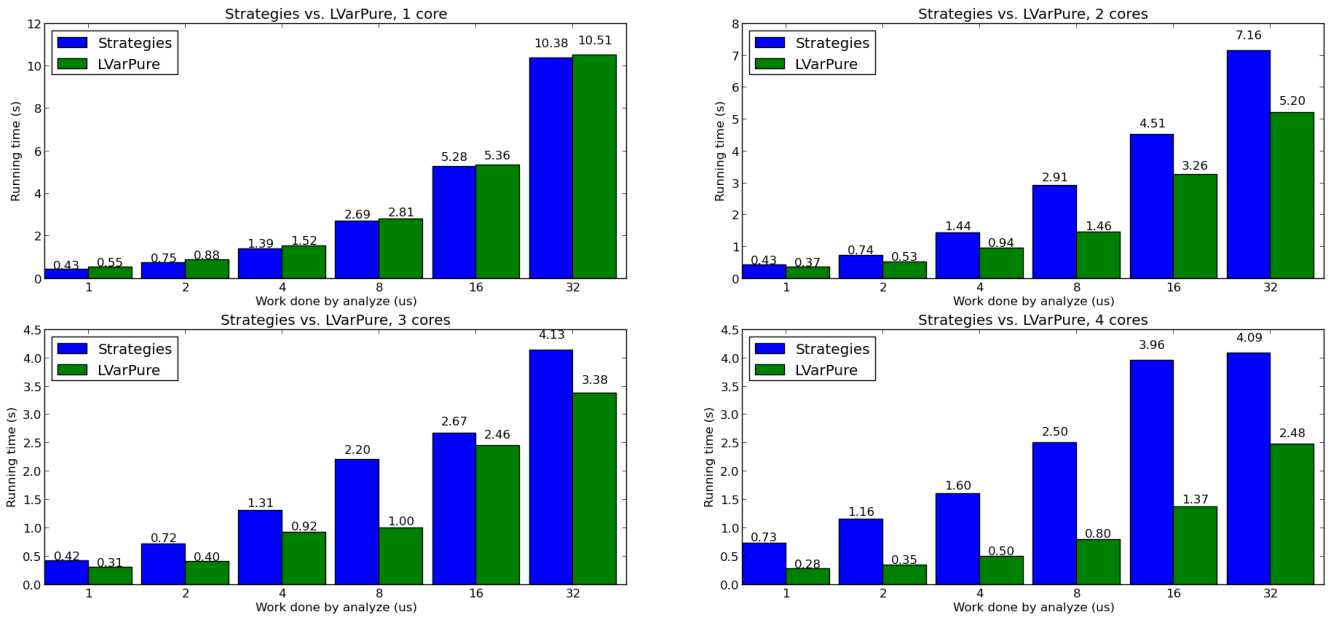

Figure 9. Running time comparison of Strategies-based and LVar-based implementations of `bf_traverse`, running on 1, 2, 3, and 4 cores on an Intel Xeon Core i5 3.1GHz (smaller is better).

ure 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 `l_acc` accumulator variable in the `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 *probation*, to D. The lattice defined by the relation $\sqsubseteq$ is extended thus:

$$\begin{aligned} &\mathit{probation} \sqsubseteq \top \\ &\forall d \in D.\ d \not\sqsubseteq \mathit{probation} \end{aligned}$$

We then propose a new operation, `consume`, that takes a pointer to an LVar l, updates the store, setting l's state to *probation*, and returns a singleton set containing the *exact* previous state of l, rather than a lower bound on that state. The idea is to ensure that, after a `consume`, any further operations on l will go awry: `put` operations will attempt to move the state of l to $\top$, resulting in **error**.

In the following example program, we use `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.

$$\begin{array}{l} \texttt{let } \mathit{cnt} = \texttt{new in} \\ \quad \texttt{let } \mathit{sum} = \texttt{new in} \\ \quad\quad \texttt{let par } p_1 = (\texttt{bump}_3\ \mathit{sum};\ \texttt{put}\ \{a\}\ \mathit{cnt}) \\ \quad\quad\quad\quad\ \ p_2 = (\texttt{bump}_4\ \mathit{sum};\ \texttt{put}\ \{b\}\ \mathit{cnt}) \\ \quad\quad\quad\quad\ \ p_3 = (\texttt{bump}_5\ \mathit{sum};\ \texttt{put}\ \{c\}\ \mathit{cnt}) \\ \quad\quad\quad\quad\ \ r\ \ = (\texttt{get}\ \mathit{cnt}\ \{a,b,c\};\ \texttt{consume}\ \mathit{sum}) \\ \quad\quad \texttt{in } \ldots\ r\ \ldots \end{array}$$

(Example 3)

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

[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 synchronization mechanism, ensuring that all increments are complete before the value is read. Three writers and one reader execute in parallel, and only when all writers complete does the reader return the sum, which in this case will be $3 + 4 + 5 = 12$.

The good news is that (Example 3) is deterministic; it will always return the same value in any execution. However, the `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` $cnt\ \{a, b, c\}$ is replaced by `get` $cnt\ \{a, b\}$. This version would have insufficient synchronization. The program could run correctly many times—if the `bump`s happen to complete before the `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 rerunning in a debug mode that is guaranteed to find a valid execution if it exists, or by using a *data-race detector* which will reproduce all races in the execution in question. In fact, the simplest form of error handling is to simply retry (or rerun from the last snapshot)—most data races make it into production only because they occur *rarely*. We have implemented a proof-of-concept interpreter and data-race detector for λ_{LVar} extended with `consume`, available 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 capability for generating unique IDs—can be implemented in λ_{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.[13] With this encoding, incrementing a shared variable l requires `put` $l\ \{\alpha\}$, where $\alpha \in \{s_1, s_2, s_3, \ldots\}$ and α has not previously been used. Rather than requiring the programmer to be responsible for creating a unique α for each parallel contribution to the counter, though, we would like to be able to provide a language construct `unique` that, when evaluated, returns a singleton set containing a single unique element of the alphabet: $\{\alpha\}$. The expression `bump` l could then simply desugar to `put` l `unique`.

Fortunately, `unique` is implementable: well-known techniques exist for generating a unique (but schedule-invariant and deterministic) identifier for a given point in a parallel execution. One such technique is to reify the position of an operation inside a tree (or DAG) of parallel evaluations. The Cilk Plus parallel programming language refers to this notion as the *pedigree* of an operation and uses it to seed a deterministic parallel random number generator [20].

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

```
let sum = new in
  let par p1 = (put sum unique; put sum unique)
          p2 = (put sum unique)
  in ...
```

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

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

$$
\begin{aligned}
[\![\texttt{unique}]\!] &= \lambda p.\ \texttt{convert}\ p \\
[\![v]\!] &= \lambda p.\ v \\
[\![Q]\!] &= \lambda p.\ Q \\
[\![\lambda v.\ e]\!] &= \lambda p.\ \lambda v.\ [\![e]\!] \\
[\![\texttt{new}]\!] &= \lambda p.\ \texttt{new} \\
[\![e_1\ e_2]\!] &= \lambda p.\ (([\![e_1]\!]\ L{:}p)\ ([\![e_2]\!]\ R{:}p)\ J{:}p) \\
[\![\texttt{put}\ e_1\ e_2]\!] &= \lambda p.\ \texttt{put}\ ([\![e_1]\!]\ L{:}p)\ ([\![e_2]\!]\ R{:}p) \\
[\![\texttt{get}\ e_1\ e_2]\!] &= \lambda p.\ \texttt{get}\ ([\![e_1]\!]\ L{:}p)\ ([\![e_2]\!]\ R{:}p) \\
[\![\texttt{convert}\ e]\!] &= \lambda p.\ \texttt{convert}\ ([\![e]\!]\ p)
\end{aligned}
$$

Figure 10. Rewrite rules for desugaring λ_{LVar} expressions with the `unique` construct to plain λ_{LVar} expressions without `unique`. Here we use "L:", "R:", and "J:" to *cons* onto the front of a list that represents a path within a fork/join DAG. These prefixes mean, respectively, "left branch", "right branch", or "after the join" of the two branches. This transformation requires a λ-calculus encoding of lists, as well as a definition of `convert` that is an injective function from these list values onto the alphabet of unique symbols.

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 λ_{LVar}'s semantics would involve parameterizing the type system by the user-specified lattice.

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

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

λ_{LVar}. CR could be used to model similar types of data structures—if versioned variables used least upper bound as their merge function for conflicts—but effects would only become visible at the end of parallel regions, rather than λ_{LVar}'s asynchronous communication within parallel regions.

Bloom and Bloom^L In the distributed systems literature, *eventually consistent* systems [28] leverage the idea of monotonicity to guarantee that, for instance, nodes in a distributed database eventually agree. The Bloom language for distributed database programming [3] guarantees eventual consistency for distributed data collections that are updated monotonically. The initial formulation of Bloom had a notion of monotonicity based on *set containment*, analogous to the store ordering for single-assignment languages given in Definition 4. However, recent work by Conway *et al*. [10] generalizes Bloom to a more flexible lattice-parameterized system, Bloom^L, in a manner analogous to our generalization from IVars to LVars. Bloom^L comes with a library of built-in lattice types and also allows for users to implement their own lattice types as Ruby classes. Although Conway *et al*. do not give a proof of eventual consistency for Bloom^L, our determinism result for λ_{LVar} suggests that their generalization is indeed safe. Moreover, although the goals of Bloom^L differ from those of LVars, we believe that Bloom^L bodes well for programmers' willingness to use lattice-based data structures, and lattice-parameterized languages based on them, to address real-world programming challenges.

9. Conclusion

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

Acknowledgments

Thanks to Aaron Turon, Neel Krishnaswami, Amr Sabry, and Chung-chieh Shan for many insightful conversations and for their expert feedback at various stages of this work. Thanks also to Amal Ahmed for originally pointing out the similarity between the Independence Lemma and the frame rule. This research was funded in part by NSF grant CCF-1218375.

References

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

[2] The Parallel Boost Graph Library. URL `http://www.boost.org/doc/libs/1_53_0/libs/graph_parallel`.

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

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

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

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

[7] 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. URL `http://dl.acm.org/citation.cfm?id=1938482.1938486`.

[8] S. Burckhardt, A. Baldassin, and D. Leijen. Concurrent programming with revisions and isolation types. In *OOPSLA*, 2010.

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

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

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

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

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

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

[15] 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.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

A. Safe Renaming

When λ_{LVar} 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 $\langle S;\ e_1\rangle \longhookrightarrow \langle S_1;\ e_1'\rangle$, generates a name that is not yet used on either side of the split and substitutes that name into $\langle S_1;\ e_1'\rangle$ in place of l.[14] We arbitrarily choose to rename locations created during the reduction of $\langle S;\ e_1\rangle$, but it would work just as well to rename those created during the reduction of $\langle S;\ e_2\rangle$.

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

$$\begin{array}{lcl} rename(\cdot,\cdot,\cdot) & : & \sigma \times S \times S \to \sigma \\ rename(\langle S';\ e\rangle, S'', S) & \triangleq & \langle S';\ e\rangle[l_1 := l_1'] \ldots [l_n := l_n'] \end{array}$$

where:

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

However, property (1) means that we cannot allow α-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 $dom(S)$.

Definition 8. A *safe renaming of* $\langle S';\ e'\rangle$ *with respect to* $\langle S;\ e\rangle \longhookrightarrow \langle S';\ e'\rangle$ 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 $dom(S') - dom(S)$, and the names $l_1', \ldots, l_n'$ that are replacing $l_1, \ldots, l_n$ do not appear in $dom(S')$.

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

If $\langle S'';\ e''\rangle$ is a safe renaming of $\langle S';\ e'\rangle$ with respect to $\langle S;\ e\rangle \longhookrightarrow \langle S';\ e'\rangle$, then S'' is by definition non-conflicting with $\langle S;\ e\rangle \longhookrightarrow \langle S';\ e'\rangle$.

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 $\langle S;\ e\rangle \longhookrightarrow \langle S';\ e'\rangle$.

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

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

$\langle S;\ e\rangle \longhookrightarrow$

$\langle S_{oldlocs}[l_1' \mapsto S'(l_1)] \ldots [l_n' \mapsto S'(l_n)];\ e'[l_1 := l_1'] \ldots [l_n := l_n']\rangle$

($\neq$ **error***),*

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

Proof sketch. By induction on the derivation of $\langle S;\ e\rangle \longhookrightarrow \langle S';\ e'\rangle$, 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* $\langle S;\ e\rangle \longhookrightarrow \langle S';\ e'\rangle$ *(where* $\langle S';\ e'\rangle \neq$ **error***) and* $S'' \neq \top_S$*, then:*

$\langle S;\ e\rangle \longhookrightarrow rename(\langle S';\ e'\rangle, S'', S)$.

Proof sketch. Straightforward application of Lemma 8. □