A Lattice-Theoretical Approach to Deterministic Parallelism with Shared State

Lindsey Kuper Ryan R. Newton
Indiana University
{lkuper, rrnewton}@cs.indiana.edu

Abstract

We present 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 partial order. Our model achieves determinism by using a novel shared data structure with an API that allows only monotonic writes and restricted reads. We give a proof of determinism for our model and show that it is expressive enough to subsume diverse existing deterministic parallel models, while providing a sound theoretical foundation for exploring controlled nondeterminism.

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

General Terms Design, Languages, Theory

Keywords Deterministic parallelism, shared state

1. Introduction

Append-only logs and monotonically increasing counters are examples of data structures in which mutators may only add information, never destroy it. These structures are a key tool for dealing with concurrent modification of data in computer systems. In a distributed computing context, publish-subscribe feeds and immutable key-value stores also leverage the principle of monotonicity.

Because state modifications that only add information can be structured to commute with one another (and avoid infamous data-race nondeterminism), such data structures are particularly useful in the context of deterministic-by-construction parallel programming models. Yet there is little in the way of a general theory of such data structures or their role in parallel programming. In this paper we take a step towards such a theory.

We begin with an example. Consider the following program, written in a hypothetical programming language with locations, standard get and put operations on locations, and a let par form for parallel evaluation of multiple subexpressions:

[Copyright notice will appear here once 'preprint' option is removed.]

$$\begin{array}{l}\mathsf{let}\ \_ = \mathsf{put}\ l\ 3\ \mathsf{in}\\ \quad \mathsf{let\ par}\ v = \mathsf{get}\ l\\ \qquad\qquad \_ = \mathsf{put}\ l\ 4\\ \quad \mathsf{in\ print}\ v\end{array} \qquad \text{(Example 1)}$$

Depending on whether get l or put l 4 executes first, the value of v printed by the last line of the program might be either 3 or 4. Hence Example 1 is nondeterministic: multiple runs of the program can produce different observable results based on choices made by the scheduler.

A straightforward modification we can make to our hypothetical language to enforce determinism is to require that variables may be written to at most once, resulting in a *single-assignment* language [24]. Such single-assignment variables are sometimes known as *IVars* because they are a special case of *I-structures* [3]—namely, those with only one cell. In a language with IVars, the second call to put would raise a run-time error, and the resulting program, since it would always produce the error, would be deterministic.

IVars enforce determinism by restricting the *writes* that can occur to a variable. However, the single-write restriction can be relaxed as long as *reads* are restricted. Suppose we modify get to take an extra argument, representing the *minimum value* that we are interested in reading from v:

$$\begin{array}{l}\mathsf{let}\ \_ = \mathsf{put}\ l\ 3\ \mathsf{in}\\ \quad \mathsf{let\ par}\ v = \mathsf{get}\ l\ 4\\ \qquad\qquad \_ = \mathsf{put}\ l\ 4\\ \quad \mathsf{in\ print}\ v\end{array} \qquad \text{(Example 2)}$$

Here, if the value of l has not yet reached 4 at the time that get l 4 is ready to run, the operation *blocks* until it does, giving put l 4 an opportunity to run first. Assuming (as we do) that the scheduler will eventually decide to run both branches of the let par expression, Example 2 is deterministic and will always result in 4 being printed. Moreover, if we had written get l 5 instead of get l 4, the program would be guaranteed to block forever.

But what if multiple subcomputations were writing to l in parallel, all with values greater than l's current value? For instance, consider this nondeterministic program:

$$\begin{array}{l}\mathsf{let}\ \_ = \mathsf{put}\ l\ 3\ \mathsf{in}\\ \quad \mathsf{let\ par}\ v = \mathsf{get}\ l\ 4\\ \qquad\qquad \_ = \mathsf{put}\ l\ 4\\ \qquad\qquad \_ = \mathsf{put}\ l\ 5\\ \quad \mathsf{in\ print}\ v\end{array} \qquad \text{(Example 3)}$$

Here, competing puts land us back where we started—Example 3 might print either 4 or 5. Rather than returning to the IVar model, though, we can regain determinism by making another change to the semantics of get: we say that if the minimum value specified

by a get operation has been reached, then the get operation returns *that minimum value*. Therefore the expression get l 4 blocks until 4 *or anything greater* has been reached, then returns 4, and Example 3 becomes deterministic. This get restriction is not as draconian as it may seem; later we will see how the total order above can be relaxed to a partial order, and *sets* of minimum values may be queried. But even with restricted gets, arbitrary puts cannot be allowed. If values both above and below a get's minimum threshold are put, then the whims of the scheduler again determine the final value, and therefore whether the get returns.

Our solution is for put to allow only *monotonically increasing* writes. Under this semantics, an attempt to write to a location l a new value that is less than l's current value simply leaves l unchanged. In the case of Example 3, either ordering of put l 5 and put l 4 will result in 5 being stored in l. Together, monotonically increasing puts and minimum-value gets yield a *deterministic-by-construction* model, guaranteeing that Example 3 and every other program written using the model will behave deterministically.

By moving from single writes and unlimited reads to multiple, monotonically increasing writes and minimum-value reads, we have generalized IVars to a data structure we call *LVars*, thus named because their states can be represented as elements of a user-specified partially ordered set that forms a *bounded join-semilattice*. This user-specified partially ordered set, which we call a *domain*, determines the semantics of the put and get operations that comprise the interface to LVars: put operations can only change an LVar's state in a way that is monotonically increasing with respect to the partial order, and get operations only allow limited observations of its state. In Example 3, for instance, the domain that determines the semantics of put and get might be the natural numbers ordered by $\leq$.

With LVars, we are able to maintain determinism even in a parallel shared-state setting, and without resorting to single assignment. Moreover, the LVar model is general enough to subsume the IVar model—as well as other deterministic parallel models—because it is parameterized by the choice of domain. Different instantiations of the domain result in a family of parallel languages that are differently expressive, but all of which are deterministic.

We make the following contributions:

- We introduce LVars as the basic building block of a model of deterministic parallelism (Section 2) and use them to define λpar, a parallel calculus with shared state based on the call-by-value λ-calculus (Section 3).
- As our main technical result, we present a proof of determinism for λpar (Section 4). A novel aspect of the proof is a "frame-rule-like" property, expressed by the Independence and Clash lemmas (Section 4.2), 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 demonstrate that λpar is sufficiently expressive to model two paradigms of deterministic parallel computation: shared-state, single-assignment models, exemplified by the Intel Concurrent Collections framework [7] and the `monad-par` Haskell library [18], and data-flow networks, exemplified by Kahn process networks [14] (Section 5).
- We propose an extension to the basic λpar model: destructive observations, enabling a limited form of nondeterminism that admits failures but not wrong answers (Section 6).

2. Domains, Stores, and Determinism

We take as the starting point for our work a call-by-value λ-calculus extended with a *store* and with communication primitives put and get that operate on data in the store. We call this language λpar. The class of programs that we are interested in modeling with λpar are those with explicit effectful operations on shared data structures, in which subcomputations may communicate with each other via the put and get operations.

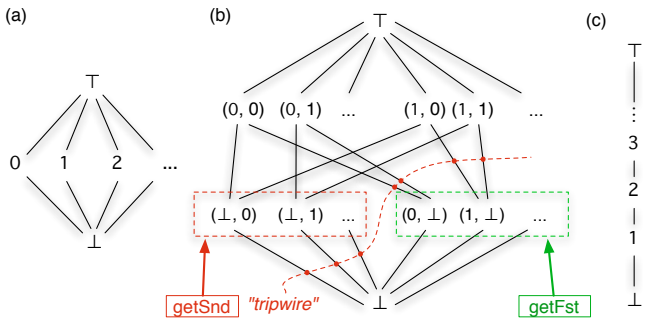

Figure 1. Example domains for common data structures: (a) IVar containing a natural number; (b) pair of natural-number-valued IVars; (c) $\leq$ ordering. Subfigure (b) is annotated with example query sets that would correspond to a blocking read of the first or second element of the pair. Any state transition crossing the "tripwire" for getSnd causes the operation to unblock and return a result.

In this setting of shared mutable state, the trick that λpar employs to maintain determinism is that stores contain *LVars*, which are a generalization of IVars [3]. Whereas IVars are single-assignment variables—either empty or filled with an immutable value—an LVar may have an arbitrary number of states forming a *domain* (or *state space*) D, which is partially ordered by a relation $\sqsubseteq$. An LVar can take on any sequence of states from the domain 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 interface presented by the get operation allows only limited observations of the LVar's state. In this section, we discuss how domains and stores work in λpar and explain how the semantics of put and get together enforce determinism in λpar programs.

2.1 Domains

The definition of λpar is parameterized by the choice of a *domain* D: to write concrete λpar programs, one must specify the domain that one is interested in working with. Therefore λpar is actually a *family* of languages, rather than a single language. Virtually any data structure to which information is added gradually can be represented as a λpar domain, including pairs, arrays, trees, maps, and infinite streams. Figure 1 gives three examples of domains for common data structures.

Formally, a domain D is a *bounded join-semilattice*.[1] In other words:

- D comes equipped with a partial order $\sqsubseteq$;
- every pair of elements in D has a least upper bound (lub) $\sqcup$;
- D has a least element $\bot$ and a greatest element $\top$.

The simplest example of a useful domain 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 domain

[1] Although we will sometimes abbreviate "bounded join-semilattice" to "lattice" for brevity's sake in the discussion that follows, λpar domains do not, in general, satisfy the properties of a lattice.

in Figure 1(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).

The motivation for requiring domains with the given structure is as follows:

- the least element, $\bot$, is needed to initialize store locations;
- the greatest element, $\top$, is needed to denote "conflicting" updates to store locations;
- the requirement that every two elements must have a lub means that it is always possible to fork a computation into subcomputations that can independently update the store and then join the results by taking the lub of updates to shared locations;
- to guarantee determinism, it is crucial that the *order* of updates by independent subcomputations is not observable. Thus direct reads that would return the "current" value of a location are not permitted. Instead, to read the contents of a location, one must provide a *query set*, which is a non-empty subset of D such that the lub of any two distinct elements of the set is $\top$. The result of reading the location bound to $d \neq \top$ is an element d' from the query set such that $d' \sqsubseteq d$. Note that d' is unique, for if there is another $d'' \neq d'$ in the query set such that $d'' \sqsubseteq d$, it would follow that $d' \sqcup d'' = d \neq \top$, which is a contradiction.

2.2 Stores

During the evaluation of a λ_{par} 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 domain 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_{par}, the type of an LVar might determine the domain of its states.

Definition 1 (store)**.** 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 $[l_1 \mapsto d_1, l_2 \mapsto d_2, \ldots]$. The state space of stores forms a bounded join-semilattice, just as D does. The least element $\bot_S$ is the empty store, and $\top_S$ is the greatest element. It is straightforward to lift the $\sqsubseteq$ and $\sqcup$ operations defined on elements of D to the level of stores:

Definition 2 (store ordering)**.** 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 (least upper bound of stores)**.** The lub of two stores S_1 and S_2 (written $S_1 \sqcup_S S_2$) is defined as follows:

- If $S_1(l) \sqcup S_2(l) = \top$ for any $l \in dom(S_1) \cap dom(S_2)$, then $S_1 \sqcup_S S_2 = \top_S$.
- 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 $[l \mapsto d_1] \sqcup_S [l \mapsto d_2] = \top_S$. Such a store can never actually arise during the execution of a λ_{par} program, because (as we will see in Section 3) any write that would take the state of l to $\top$ raises an error before the write can occur. If that were not the case, the store $\top_S$ would arise from the evaluation of a λ_{par} program if, for instance, one subcomputation wrote d_1 to l while a parallel subcomputation wrote d_2 to l.

2.3 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; it updates the store, merging the current state of the LVar with the new state by taking their lub, and pushes 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 takes a pointer to an LVar and a query set Q, and returns a singleton set containing the unique element of Q that appears *at or below* the LVar's state in the lattice. Hence get expressions allow *limited observations* of the state of an LVar.

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 $\{d\}$).

2.4 Monotonic Store Growth and Determinism

In IVar-based languages, a store can only change through the addition of new bindings, since existing bindings cannot be updated (except for, perhaps, an initial update from $\bot$ 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, the single-assignment *Featherweight CnC* language [7] 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 $S \sqsubseteq_S S'$) 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_{par}, 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_{par} 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 4 that λ_{par} 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.

Given a domain D with elements $d \in D$:

configurations	σ	::=	$\langle S;\ e\rangle \mid \textbf{error}$
expressions	e	::=	$x \mid v \mid e\ e \mid \mathsf{new} \mid$
			$\mathsf{put}\ e\ e \mid \mathsf{get}\ e\ e \mid \mathsf{convert}\ e$
values	v	::=	$l \mid Q \mid \lambda x.\, e$
query set literals	Q	::=	$\{d_1, d_2, \ldots, d_n\} \mid$
			$\{d \mid pred(d)\}$
			(where $pred(d)$ is computable)
stores	S	::=	$[l_1 \mapsto d_1, l_2 \mapsto d_2, \ldots]$

Figure 2. Syntax for λ_{par}.

3. λ_{par}: Syntax and Semantics

The syntax and operational semantics of λ_{par} appear in Figures 2 and 3, respectively. As we've noted, both the syntax and semantics are parameterized by the domain D. The operational semantics is defined on *configurations* $\langle S;\ e\rangle$ comprising a store and an expression. We identify any configuration in which $S = \top_S$ with the *error configuration*, written **error**. **error** is a unique element added to the set of configurations, but it also forms an equivalence class such that for all e, $\langle \top_S;\ e\rangle = \textbf{error}$. The metavariable σ ranges over configurations.

Figure 3 shows two disjoint sets of reduction rules: those that step to configurations other than **error**, and those that step to **error**. Most of the rules that step to **error** are merely propagating existing errors along. A new **error** can only arise by way of E-PARAPPERR, 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 the state of the location to $\top$.

3.1 Semantics of the Store Interface

The reduction rules E-NEW, E-PUTVAL, and E-GETVAL in Figure 3 respectively express the semantics of the new, put, and get operations described in Section 2.3. The intuition behind get is that it specifies a subset of the lattice that is "horizontal": no two elements in the subset can be above or below one another. This property is enforced in the E-GETVAL rule by an *incompatibility* condition on the query set Q that states that the least upper bound of any two distinct elements in Q must be $\top$.[2] Intuitively, each element in the query set is an "alarm" that detects the activation of itself or any state above it. One way of visualizing the query 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 1(b). The query set $\{(\bot, 0), (\bot, 1), ...\}$ (or a subset thereof) would pass the incompatibility test, as would the query set $\{(0, \bot), (1, \bot), ...\}$ (or a subset thereof), but a combination of the two would not pass.

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. The arguments to put and get are evaluated in arbitrary order, although not simultaneously. [3]

[2] Although $incomp(Q)$ is given as a premise of the E-GETVAL reduction rule (indicating that it is checked at runtime), in a real implementation the incompatibility condition on query sets might be checked statically, eliminating the need for the runtime check. In fact, a real implementation could forego any runtime representation of query sets.

[3] It would, however, be straightforward to add to the semantics E-PARPUT and E-PARGET rules analogous to E-PARAPP, should simultaneous evaluation of put and get arguments be desired.

3.2 Fork-Join Parallelism

λ_{par} has an explicitly parallel reduction semantics: the E-PARAPP rule in Figure 3 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 $e_1'\ e_2'$. 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 the resulting two stores are combined by taking their least upper bound when the subcomputations rejoin.[4]

Although the semantics admits such parallel reductions, λ_{par} is still call-by-value in the sense that arguments to functions 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 *parallel composition* of expressions e_1 and e_2 as follows:

$$((\lambda x.\, (\lambda y.\, e_3))\ e_1)\ e_2 \qquad \text{(Example 4)}$$

Although e_1 and e_2 are evaluated in parallel, e_3 cannot be evaluated until both e_1 and e_2 are evaluated, because the call-by-value semantics does not allow β-reduction until the operand is fully evaluated, and because it further disallows reduction under a λ-term (sometimes called "full β-reduction"). In the terminology of parallel programming, the above 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 [11].

Conversely, to *sequentially* compose e_1 before e_2 before e_3, we could write the following:

$$(\lambda x.\, ((\lambda y.\, e_3)\ e_2))\ e_1 \qquad \text{(Example 5)}$$

Sequential composition is necessary for ordering side-effecting put and get operations on the store. For that reason, full β-reduction would be a poor choice, but parallel call-by-value gives λ_{par} both sequential and parallel composition, without introducing additional language forms.

Notational shorthand For clarity, we will write $\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$ as a shorthand for $((\lambda x.\, e_2)\ e_1)$. We also use semicolon as sugar for sequential composition: for example, $e_1; e_2$ rather than $\mathsf{let}\ \_ = e_1\ \mathsf{in}\ e_2$. Finally, the following let par expression desugars to the parallel composition expression in (Example 4) and should be read as "compute e_1 and e_2 in parallel before computing e_3":

$$\begin{aligned}&\mathsf{let\ par}\ x = e_1\\ &\qquad\quad y = e_2\\ &\mathsf{in}\ e_3\end{aligned}$$

As mentioned above, 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 on the left side of Figure 4. In the terminology of parallel algorithms, the λ_{par} 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 like that shown in Figure 4 (right). This adds the ability to construct arbitrary non-series-parallel dependency graphs, just as with *first-class futures* [23].

[4] A subtle point that E-PARAPP and E-PARAPPERR must address is 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 as part of a more wide-ranging discussion in Section 4.1.

Given a domain D with elements $d \in D$, and a value-conversion function δ:

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

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

E-REFL
$$\frac{}{\langle S;\, e\rangle \longrightarrow \langle S;\, e\rangle}$$

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

E-PUT-1
$$\frac{\langle S;\, e_1\rangle \longrightarrow \langle S_1;\, e_1'\rangle}{\langle S;\, \mathsf{put}\ e_1\ e_2\rangle \longrightarrow \langle S_1;\, \mathsf{put}\ e_1'\ e_2\rangle}$$

E-PUT-2
$$\frac{\langle S;\, e_2\rangle \longrightarrow \langle S_2;\, e_2'\rangle}{\langle S;\, \mathsf{put}\ e_1\ e_2\rangle \longrightarrow \langle S_2;\, \mathsf{put}\ e_1\ e_2'\rangle}$$

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

E-GET-1
$$\frac{\langle S;\, e_1\rangle \longrightarrow \langle S_1;\, e_1'\rangle}{\langle S;\, \mathsf{get}\ e_1\ e_2\rangle \longrightarrow \langle S_1;\, \mathsf{get}\ e_1'\ e_2\rangle}$$

E-GET-2
$$\frac{\langle S;\, e_2\rangle \longrightarrow \langle S_2;\, e_2'\rangle}{\langle S;\, \mathsf{get}\ e_1\ e_2\rangle \longrightarrow \langle S_2;\, \mathsf{get}\ e_1\ e_2'\rangle}$$

E-GETVAL
$$\frac{S(l) = d_2 \qquad incomp(Q) \qquad Q \subseteq D \qquad d_1 \in Q \qquad d_1 \sqsubseteq d_2}{\langle S;\, \mathsf{get}\ l\ Q\rangle \longrightarrow \langle S;\, \{d_1\}\rangle}$$

E-CONVERT
$$\frac{\langle S;\, e\rangle \longrightarrow \langle S';\, e'\rangle}{\langle S;\, \mathsf{convert}\ e\rangle \longrightarrow \langle S';\, \mathsf{convert}\ e'\rangle}$$

E-CONVERTVAL
$$\frac{}{\langle S;\, \mathsf{convert}\ v\rangle \longrightarrow \langle S;\, \delta(v)\rangle}$$

E-BETA
$$\frac{}{\langle S;\, (\lambda x.\, e)\ v\rangle \longrightarrow \langle S;\, e[x := v]\rangle}$$

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

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

E-REFLERR
$$\frac{}{\textbf{error} \longrightarrow \textbf{error}}$$

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

E-APPERR-1
$$\frac{\langle S;\, e_1\rangle \longrightarrow \textbf{error}}{\langle S;\, e_1\ e_2\rangle \longrightarrow \textbf{error}}$$

E-APPERR-2
$$\frac{\langle S;\, e_2\rangle \longrightarrow \textbf{error}}{\langle S;\, e_1\ e_2\rangle \longrightarrow \textbf{error}}$$

E-PUTERR-1
$$\frac{\langle S;\, e_1\rangle \longrightarrow \textbf{error}}{\langle S;\, \mathsf{put}\ e_1\ e_2\rangle \longrightarrow \textbf{error}}$$

E-PUTERR-2
$$\frac{\langle S;\, e_2\rangle \longrightarrow \textbf{error}}{\langle S;\, \mathsf{put}\ e_1\ e_2\rangle \longrightarrow \textbf{error}}$$

E-PUTVALERR
$$\frac{S(l) = d_2 \qquad d_1 \in D \qquad d_1 \sqcup d_2 = \top}{\langle S;\, \mathsf{put}\ l\ \{d_1\}\rangle \longrightarrow \textbf{error}}$$

E-GETERR-1
$$\frac{\langle S;\, e_1\rangle \longrightarrow \textbf{error}}{\langle S;\, \mathsf{get}\ e_1\ e_2\rangle \longrightarrow \textbf{error}}$$

E-GETERR-2
$$\frac{\langle S;\, e_2\rangle \longrightarrow \textbf{error}}{\langle S;\, \mathsf{get}\ e_1\ e_2\rangle \longrightarrow \textbf{error}}$$

E-CONVERTERR
$$\frac{\langle S;\, e\rangle \longrightarrow \textbf{error}}{\langle S;\, \mathsf{convert}\ e\rangle \longrightarrow \textbf{error}}$$

Figure 3. An operational semantics for λ_{par}.

3.3 Programming with put and get

For our first example of a λ_{par} program, we choose our domain to be pairs of natural-number-valued IVars, represented by the lattice shown in Figure 1(b). With D instantiated thusly, we can write the following program:

$$\begin{array}{l}\mathsf{let}\ p = \mathsf{new}\ \mathsf{in}\\ \quad \mathsf{let}\ \_ = \mathsf{put}\ p\ \{(3,4)\}\ \mathsf{in}\\ \quad\quad \mathsf{let}\ v_1 = \mathsf{get}\ p\ \{(\bot, n) \mid n \in \mathbb{N}\}\ \mathsf{in}\\ \quad\quad\quad \ldots \mathsf{convert}\ v_1 \ldots\end{array} \qquad \text{(Example 6)}$$

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 query 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 the lattice $(D, \sqsubseteq)$. In particular, the pair $(\bot, 4)$ is a member of Q, and $(\bot, 4) \sqsubseteq (3,4)$ in $(D, \sqsubseteq)$. Therefore, get $p\ \{(\bot, n) \mid n \in \mathbb{N}\}$ returns the singleton set $\{(\bot, 4)\}$, which is a first-class value in λ_{par} that can, for example, subsequently be passed to put.

Since query sets can be cumbersome to read, we can define some convenient shorthands getFst and getSnd for working with the domain of pairs:

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

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

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 6) 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 green- and red-boxed elements of the lattice in Figure 1(b) to get as a single query set, which would fail the incompatibility criterion.

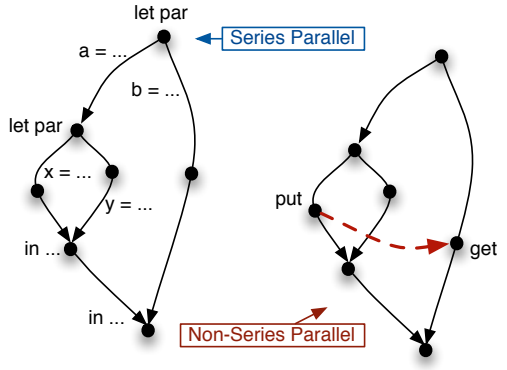

Figure 4. A series-parallel graph induced by basic parallel λ-calculus evaluation (left) vs. a non-series-parallel graph created by put/get communication (right).

Blocking reads On the other hand, consider the following:

$$\begin{array}{l}\mathsf{let}\ p = \mathsf{new}\ \mathsf{in}\\ \quad \mathsf{let}\ \_ = \mathsf{put}\ p\ \{(\bot, 4)\}\ \mathsf{in}\\ \quad\quad \mathsf{let}\ \mathsf{par}\ v_1 = \mathsf{getFst}\ p \qquad \text{(Example 7)}\\ \quad\quad\quad\quad\ \ \_\ = \mathsf{put}\ p\ \{(3, 4)\}\\ \quad\quad \mathsf{in}\ \ldots\mathsf{convert}\ v_1 \ldots\end{array}$$

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 7), although the query set $\{(n, \bot) \mid n \in \mathbb{N}\}$ is incompatible, the E-GETVAL rule cannot apply because there is no state in the query set that is lower than the state of p in the lattice—that is, we are trying to get something that isn't yet there! It is only after p's state is updated that the premise is satisfied and the rule applies.

3.4 Converting from Query Sets to λ-terms and Back

There are two different worlds that λ_{par} values may inhabit: the world of query sets, and the world of λ-terms. But if these worlds are disjoint—if query set values are opaque atoms—certain programs are impossible to write. For example, implementing single-assignment arrays in λ_{par} requires that arbitrary array indices can be computed and converted to query sets.

Thus we parameterize our semantics by a *conversion function*, $\delta : v \to v$, to which λ_{par} provides an interface through its convert language form. The conversion function can arbitrarily convert between representations of values as query sets and representations as λ-terms. It is *optional* in the sense that providing an identity or empty function is acceptable, and leaves λ_{par} sensible but less expressive (*i.e.*, query sets are still first-class values, but usable only for passing to get and put).

The two most common applications of value conversion are *reification* and *reflection*. Reification is the process of converting a query set to a λ-term. A typical reification function might, for instance, convert $\{(3, 4)\}$ to a standard Church encoding for pairs of natural numbers. The last lines of (Example 6) and (Example 7) illustrate this use of convert—in both cases, convert takes the query set returned from a call to get and converts it into a form usable by the rest of the program.[5] Conversely, reflection constructs query sets Q from a λ-term representation. Because of the general $v \to v$ signature of the conversion function, it is possible to use one δ for both reification and reflection. It would also be possible to parameterize the semantics by an arbitrary number of different functions, but doing so would not gain generality (and would clutter the determinism proof with additional boilerplate).

Location hygiene restriction There are some restrictions on valid δ functions. Namely, its domain and range must not contain locations l, and must contain only closed λ-terms. For example, $\delta(l)$ is required to be undefined, and any term $\delta(val) = \lambda v.\, e$ must not have any locations l in e. The reason for this is that we depend on being able to rename locations, and thus values may not hard-code a dependence on particular locations.

[5] In fact, a reasonable alternative definition of λ_{par} would remove query set values entirely and require that query set inputs and outputs to get/put be implicitly converted. Yet the language is deterministic even in its more general form—with first-class query sets—and we do not want to unduly restrict the language.

$$\text{E-APP-1}\quad \frac{\langle S;\ e_1\rangle \longrightarrow \langle S_1;\ e_1'\rangle}{\langle S;\ e_1\ e_2\rangle \longrightarrow \langle S_1;\ e_1'\ e_2\rangle} \qquad \text{E-APP-2}\quad \frac{\langle S;\ e_2\rangle \longrightarrow \langle S_2;\ e_2'\rangle}{\langle S;\ e_1\ e_2\rangle \longrightarrow \langle S_2;\ e_1\ e_2'\rangle}$$

Figure 5. An alternate operational semantics for λ_{par}: remove the reflexive reduction rules E-REFL and E-REFLERR from the semantics of Figure 3, and add the two rules shown here.

$$\text{E-APP-1}\quad \frac{\langle S;\ e_1\rangle \longrightarrow \langle S_1;\ e_1'\rangle}{\langle S;\ e_1\ v\rangle \longrightarrow \langle S_1;\ e_1'\ v\rangle} \qquad \text{E-APP-2}\quad \frac{\langle S;\ e_2\rangle \longrightarrow \langle S_2;\ e_2'\rangle}{\langle S;\ v\ e_2\rangle \longrightarrow \langle S_2;\ v\ e_2'\rangle}$$

$$\text{E-GETVALBLOCK}\quad \frac{S(l) = d_2 \qquad incomp(Q) \qquad Q \subseteq D \qquad Q \neq \emptyset \qquad \forall\, d_1 \in Q.\, d_1 \not\sqsubseteq d_2}{\langle S;\ \mathsf{get}\ l\ Q\rangle \longrightarrow \langle S;\ \mathsf{get}\ l\ Q\rangle}$$

Figure 6. A further tweak to the operational semantics of Figure 5.

3.5 Running Our Research

In addition to the version of λ_{par} presented in this paper, we have developed a runnable model of a variant of λ_{par} using the PLT Redex semantics engineering toolkit [10], with the domain D instantiated to $(\{\top, \bot\} \cup \mathbb{N}, \leq)$, where $\leq$ is defined in the usual way for natural numbers. Our Redex model and test suite are available at `https://github.com/lkuper/lambdapar-redex`.

The reduction relation given in the Redex model omits the reduction rules E-REFL and E-REFLERR shown in the semantics of Figure 3 and adds two rules, E-APP-1 and E-APP-2, shown in Figure 5, by which parallel application expressions may take a step even if only one of their subexpressions can take a step. This version explores all interleavings of parallel computations, but avoids confounding Redex with reflexive reductions.

It is also convenient to have a "quick mode" that only samples a single execution order. To that end we define in Redex a third version of the reduction relation, in which we replace the E-APP-1 and E-APP-2 rules of Figure 5 with the more restricted versions shown in Figure 6, in which the subexpression that is not taking a step must be a *value*, and add the E-GETVALBLOCK rule of Figure 6, which allows a *blocked* get expression to step to itself. A detailed discussion of the impetus behind the tweaked semantics of Figures 5 and 6 is beyond the scope of this paper, but is available in the documentation provided with our Redex model.

4. Proof of Determinism for λ_{par}

Our main technical result is a proof of determinism for the λ_{par} language. Most proofs are elided or only sketched here; the complete proofs appear in the companion technical report [15].

4.1 Framing and Renaming

Figure 7 shows a *frame rule*, due to O'Hearn *et al.* [20], which captures the idea that, given a program c with precondition p that holds before it runs and postcondition q that holds afterward, some disjoint condition r should hold both before and after c runs. If we think of c as a put operation, and of p and q as, respectively, the state of a store before and after the put occurs, then the *local reasoning* principle we get from such a rule is that we need only worry about the piece of the store that the put affects (represented by p and q), and not the rest (represented by r). In the setting of a single-assignment language, it is easy to see that such a local reasoning property holds: a put cannot affect any store locations that have already been written to, because multiple writes are not allowed. For λ_{par}, we can still state a property that is not entirely

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 2 (Independence), simplified:

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

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

unlike a frame rule, but to do so we have to define a notion of *non-conflicting* stores.

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

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

Requiring that a store S'' be non-conflicting with a transition $\langle S;\ e\rangle \hookrightarrow \langle S';\ e'\rangle$ is not as strict 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 $\langle S;\ e\rangle \hookrightarrow \langle S';\ e'\rangle$, 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 $dom(S)$.

Renaming Recall that when λ_{par} programs fork into two subcomputations via the E-ParApp rule, the subcomputations' stores are merged (via the lub operation) when they join again. Therefore we need to ensure that the following two properties hold:

1. Location names created before the fork still match up with each other after the join.
2. Location names created by each subcomputation *during* the fork (that is, during the time the subcomputations are running independently) do *not* match up with each other accidentally—*i.e.*, they do not collide.

Point (2) is why it is necessary to *rename* locations in the E-ParApp (and E-ParAppErr) rule. This is accomplished by the $rename(\langle S_1;\ e_1'\rangle, S_2, S)$ metafunction, which has the following semantics: for each location in the set $dom(S_1) - dom(S)$, generate a fresh location name not in the set $dom(S_2) - dom(S)$, capture-avoidingly substitute it into the configuration $\langle S_1;\ e_1'\rangle$, and return the resulting configuration.[6]

Yet point (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 in the context of a *transition* from configuration to configuration. Lemma 1 expresses the circumstances under which renaming is permitted. It says that fresh locations $l_1, \ldots, l_n$ created *during* a reduction step can be given new names $l_1', \ldots l_n'$ that do not appear in the domain of the original store, as long as we also consistently rename locations in the expression being reduced.[7]

Lemma 1 (Renaming of Fresh Locations)**.** *If* $\langle S;\ e\rangle \hookrightarrow \langle S'[l_1 \mapsto d_1] \ldots [l_n \mapsto d_n];\ e'\rangle$*, where* $dom(S) = dom(S')$ *and* $l_i \notin dom(S)$ *for all* $1 \le i \le n$*, then:*

For all $l_1' \ldots l_n'$ *such that* $l_i' \notin dom(S)$ *and* $l_i' \neq l_j$ *for distinct* i, j *for all* $1 \le j \le n$*:*

$\langle S;\ e\rangle \hookrightarrow \langle S'[l_1' \mapsto d_1] \ldots [l_n' \mapsto d_n];\ e'[l_1 := l_1'] \ldots [l_n := l_n']\rangle$.

4.2 Supporting Lemmas

Lemmas 2, 3, and 4 express three key properties that we need for establishing determinism. Lemma 2 expresses our local reasoning property: it says that if a transition steps from $\langle S;\ e\rangle$ 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), will step to $\langle S' \sqcup_S S'';\ e'\rangle$. The only restrictions on S'' are that $S' \sqcup_S S''$ cannot be $\top_S$, and that S'' must be non-conflicting with the original transition $\langle S;\ e\rangle \hookrightarrow \langle S';\ e'\rangle$.

Lemma 3 handles the case where $S' \sqcup_S S'' = \top_S$ and ensures that in that case, $\langle S \sqcup_S S'';\ e\rangle$ steps to **error**. In either case, whether the transition results in $\langle S' \sqcup_S S'';\ e'\rangle$ or in **error**, we know that it will never result in a configuration containing some other $e'' \neq e'$. Finally, Lemma 4 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 2 (Independence)**.** *If* $\langle S;\ e\rangle \hookrightarrow \langle S';\ e'\rangle$ *(where* $\langle S';\ e'\rangle \neq$ **error***), then for all* S'' *such that* S'' *is non-conflicting with* $\langle S;\ e\rangle \hookrightarrow \langle S';\ e'\rangle$ *and* $S' \sqcup_S S'' \neq \top_S$*:*

$\langle S \sqcup_S S'';\ e\rangle \hookrightarrow \langle S' \sqcup_S S'';\ e'\rangle$.

Proof sketch. By induction on the derivation of $\langle S;\ e\rangle \hookrightarrow \langle S';\ e'\rangle$, by cases on the last rule in the derivation. The requirement that S'' is non-conflicting with $\langle S;\ e\rangle \hookrightarrow \langle S';\ e'\rangle$ is only needed in the E-New case. □

Lemma 3 (Clash)**.** *If* $\langle S;\ e\rangle \hookrightarrow \langle S';\ e'\rangle$ *(where* $\langle S';\ e'\rangle \neq$ **error***), then for all* S'' *such that* S'' *is non-conflicting with* $\langle S;\ e\rangle \hookrightarrow \langle S';\ e'\rangle$ *and* $S' \sqcup_S S'' = \top_S$*:*

$\langle S \sqcup_S S'';\ e\rangle \hookrightarrow$ **error**.

Proof sketch. By induction on the derivation of $\langle S;\ e\rangle \hookrightarrow \langle S';\ e'\rangle$, by cases on the last rule in the derivation. As with Lemma 2, the requirement that S'' is non-conflicting with $\langle S;\ e\rangle \hookrightarrow \langle S';\ e'\rangle$ is only needed in the E-New case. □

Lemma 4 (Error Preservation)**.** *If* $\langle S;\ e\rangle \hookrightarrow$ **error** *and* $S \sqsubseteq_S S'$*, then* $\langle S';\ e\rangle \hookrightarrow$ **error**.

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

4.3 Diamond Lemma

Lemma 5 does the heavy lifting of our determinism proof: it establishes the *diamond property* (or *Church-Rosser property* [4]), which says that if a configuration steps to two different configurations, there exists a single third configuration to which those configurations both step.

Lemma 5 (Diamond)**.** *If* $\sigma \hookrightarrow \sigma_a$ *and* $\sigma \hookrightarrow \sigma_b$*, then there exists* σ_c *such that* $\sigma_a \hookrightarrow \sigma_c$ *and* $\sigma_b \hookrightarrow \sigma_c$.

[6] We arbitrarily choose to rename locations created during the reduction of $\langle S;\ e_1\rangle$; it would work just as well to rename those created during the reduction of $\langle S;\ e_2\rangle$.

[7] Since λ_{par} locations are drawn from a distinguished set *Loc*, they cannot occur in the user's domain D—that is, locations in λ_{par} may not contain pointers to other locations. This makes the substitution in Lemma 1 safe.

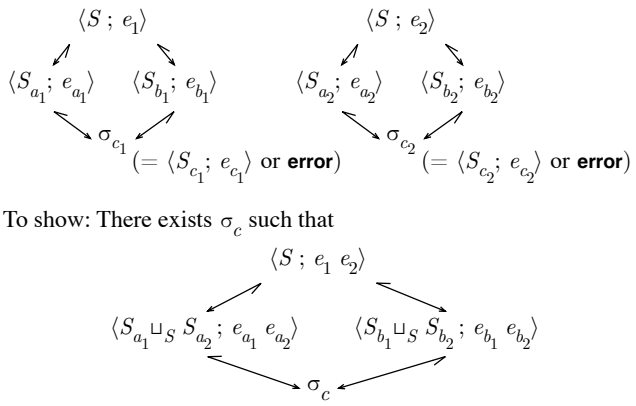

Figure 8. Diagram of the subcase of Lemma 5 in which the E-PARAPP rule is the last rule in the derivation of both $\sigma \hookrightarrow \sigma_a$ and $\sigma \hookrightarrow \sigma_b$. We are required to show that, if the configuration $\langle S;\ e_1\ e_2\rangle$ steps by the E-PARAPP reduction rule to two different configurations, $\langle S_{a_1} \sqcup_S S_{a_2};\ e_{a_1}\ e_{a_2}\rangle$ and $\langle S_{b_1} \sqcup_S S_{b_2};\ e_{b_1}\ e_{b_2}\rangle$, they both step to some third configuration σ_c.

Proof sketch. By induction on the derivation of $\sigma \hookrightarrow \sigma_a$, by cases on the last rule in the derivation. The most interesting subcase is that in which the E-PARAPP rule is the last rule in the derivation of both $\sigma \hookrightarrow \sigma_a$ and $\sigma \hookrightarrow \sigma_b$. Here Lemmas 2, 3, and 4 all play a role. In particular, as Figure 8 shows, appealing to the induction hypothesis alone is not enough to complete the case, and Lemmas 2 and 3 allow us to "piece together" stores that have resulted from the two branches of evaluation. It is because of the renaming occurring in E-PARAPP that we are able to satisfy the non-conflicting premise of Lemmas 2 and 3 at each point where we appeal to them. □

We can readily restate Lemma 5 as Corollary 1:

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

Proof. Choose $i = j = 1$. The proof follows immediately from Lemma 5. □

4.4 Confluence Lemmas and Determinism

With Lemma 5 in place, we can straightforwardly generalize its result to multiple steps, by induction on the number of steps, as Lemmas 6, 7, and 8 show. [8]

Lemma 6 (Strong One-Sided Confluence)**.** *If $\sigma \hookrightarrow \sigma'$ and $\sigma \hookrightarrow^m \sigma''$, where $1 \leq m$, then there exist σ_c, i, j such that $\sigma' \hookrightarrow^i \sigma_c$ and $\sigma'' \hookrightarrow^j \sigma_c$ 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 7 (Strong Confluence)**.** *If $\sigma \hookrightarrow^n \sigma'$ and $\sigma \hookrightarrow^m \sigma''$, where $1 \leq n$ and $1 \leq m$, then there exist σ_c, i, j such that $\sigma' \hookrightarrow^i \sigma_c$ and $\sigma'' \hookrightarrow^j \sigma_c$ and $i \leq m$ and $j \leq n$.*

[8] These lemmas are 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 2 through 5, but the statements and proofs of those lemmas differ considerably in our setting.

Proof sketch. By induction on n. In the base case of $n = 1$, the result is immediate from Lemma 6. □

Lemma 8 (Confluence)**.** *If $\sigma \hookrightarrow^* \sigma'$ and $\sigma \hookrightarrow^* \sigma''$, then there exists σ_c such that $\sigma' \hookrightarrow^* \sigma_c$ and $\sigma'' \hookrightarrow^* \sigma_c$.*

Proof. Strong Confluence (Lemma 7) implies Confluence. □

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

Proof. We have from Lemma 8 that there exists σ_c such that $\sigma' \hookrightarrow^* \sigma_c$ and $\sigma'' \hookrightarrow^* \sigma_c$. Since σ' and σ'' can only step to themselves, we must have $\sigma' = \sigma_c$ and $\sigma'' = \sigma_c$, hence $\sigma' = \sigma''$. □

5. Modeling Other Deterministic Parallel Models

The λ_{par} programming model is general enough to subsume two rather different families of deterministic-by-construction parallel computation models. The first category is single-assignment models, from which we'll take Intel's Concurrent Collections framework [7] and Haskell's `monad-par` library [18] as two examples. The second is data-flow networks, specifically Kahn process networks (KPNs) [14]. In Section 7, we discuss additional models that are related to, but not directly modeled by, λ_{par}.

5.1 Concurrent Collections

In Section 2.4, we mentioned the *Featherweight CnC* language and its monotonically growing memory store. *Featherweight CnC* is a simplified model of the Concurrent Collections (CnC) [7] language for composing graphs of "steps", more commonly known as *actors*, which are implemented separately in a general-purpose language (C++, Java, Haskell, or Python). To begin execution, a subset of steps are invoked at startup time. Each step, when executed, may perform puts and gets on global, shared data collections (tables of IVars), as well as send messages to invoke other steps. The steps themselves are stateless, except for the information they store externally in the aforementioned tables.

The role of monotonicity has been understood, at least informally, in the design of CnC. However, this has not—until now—led to a treatment of shared data collections as general as λ_{par}. λ_{par} subsumes CnC in the following sense. If the language used to express CnC steps is the call-by-value λ-calculus, then CnC programs can be translated to λ_{par}; each step would become a function definition, generated in the following way:

- Each step function takes a single argument (its message, or in CnC terminology, its *tag*) and returns $\{\}$—our *unit*, the empty query set—being executed for effect only.
- All invocations of other steps (message sends) within the step body, are aggregated at the end of the function and performed inside a `let par`. This is the sole source of parallelism. The aggregation can be accomplished either statically, by a program transformation that moves sends, or by dynamic buffering of the outgoing sends.
- The rest of the body of a step is translated directly: puts on data collections become λ_{par} `puts`; gets become become λ_{par} `gets`.

The following skeleton shows the form of a program converted by the above method. It first defines steps, then launches the initial batch of "messages", and finally reads whatever result is desired.

```
let step1 = λmsg. get ... ; put ... ;
                  let par _ = step1 ...
                          _ = step2 ...
                          _ = step2 ...
                  in {}
in let step2 = ...
in let data1 = new        - - global data collections
in let par _ = step1 33   - - invoke initial steps
         _ = step2 44
in convert (get data1 key)    - - retrieve final result
```

Somewhat surprisingly, the CnC programming model is *not* implementable in a parallel call-by-value λ-calculus extended only with IVars. In fact, it was this observation that began the inquiry leading to the development of λ_{par}. The reason is that CnC provides globally scoped, extensible *tables* of IVars, not just IVars alone. While a λ-calculus augmented with IVars could model shared global IVars, or even fixed collections of IVars, it is impossible to create a mutable, extensible table data structure with IVars alone. For intuition, consider implementing a trie with IVars. To add different leaves in parallel, the root of the trie must be filled—but which mutator fills it? And how, without (illegally) testing an IVar for emptiness? In short, solving this problem requires unification between partially populated structures: LVars.

Finally, if there were not already a determinism result for CnC [7], one could bootstrap determinism by proving that every valid step in a CnC semantics maps onto one or more evaluation steps for the translated version under the λ_{par} semantics; that is, the λ_{par} encoding simulates all possible executions of the CnC program, and since it yields a single answer, so does the CnC program. Moreover, because λ_{par} is simply an extension of the untyped λ-calculus, the determinism result in this paper is more readily reusable, as well as more general, than the Featherweight CnC result (which is previous work by the second author and others [7]).

5.2 The `monad-par` Haskell library

The `monad-par` package for Haskell provides a parallel deterministic programming model with an explicit fork operation together with first-class IVars. `monad-par` uses explicit sequencing via a monad, together with Haskell's lazy evaluation. To translate `monad-par` programs to λ_{par}, evaluation order can be addressed using standard techniques, and λ_{par} can model `monad-par`'s fork operation with let par, using the method in Section 3.2. But because `monad-par` has no *join* operations (IVar gets being the only synchronization mechanism), it would be necessary to use continuation-passing style in the translation. If the original `monad-par` program forks a child computation and returns, the translated program must invoke both the fork and its continuation within a let par expression.

A final wrinkle for translation of `monad-par` programs into λ_{par} is that while `monad-par` IVars may contain other IVars, LVars cannot contain LVars. This problem can be overcome by using a type-directed translation. Each IVar is represented by the wide, height-three lattice shown in Figure 1(a); therefore multiple IVars are modeled by product lattices. For example, a location of type `IVar (IVar Int, IVar Char)` in `monad-par` would correspond to a lattice similar to that pictured in Figure 1(b). Also, chaining IVar type constructors, `IVar (IVar (...))`, simply adds additional empty states (repeatedly lifting the domain with a new $\bot$). All these types create larger state spaces, but do not pose a fundamental barrier to encoding as LVars.

Although λ_{par} is a calculus rather than a practical programming language, the exercise of modeling `monad-par` in λ_{par} suggests practical extensions to `monad-par`. For example, additional data structures beyond IVars could be provided (*e.g.*, maps or tries), using the λ_{par} translation to ensure determinism is retained.

5.3 Kahn Process Networks

Data-flow models have been a topic of theoretical [14] and practical [13] study for decades. In particular, Kahn's 1974 paper crystallized the contemporary work on data-flow with a denotational account of *Kahn process networks* (KPNs)—a deterministic model in which a network of processes communicates through single-reader, single-writer FIFO channels with non-blocking writes and blocking reads. Because λ_{par} is general enough to subsume KPNs, it represents a step towards bringing the body of work on data-flow into the broader context of functional and single-assignment languages.

To map KPNs into λ_{par}, we represent FIFOs as ordered sequences of values, monotonically growing on one end (*i.e.*, channel *histories*). In fact, the original work on KPNs [14] used exactly this representation (and the complete partial order based on it) to establish determinism. However, to our knowledge neither KPNs nor any other data-flow model has generalized the data structures used for communication beyond FIFOs to include other monotonically-growing structures (*e.g.*, maps).

An LVar representing a FIFO has a state encoding all elements sent on that FIFO to date. We represent sequences as sets of $(\mathit{index}, \mathit{value})$ associations with subset inclusion as the order $\sqsubseteq$. For example, $\{(0, a), (1, b)\}$ encodes a two-element sequence. This makes it convenient to write query sets such as $\{(0, n) \mid n \in \mathbb{N}\}$, which will match any state encoding a channel history with a natural number value in position 0.

In this encoding, the producers and consumers using a FIFO must explicitly keep track of what position they read and write, *i.e.*, the "cursor". This contrasts with an imperative formulation, where advancing the cursor is a side effect of "popping" the FIFO. A proper encoding of FIFO behavior writes and reads consecutive positions only.[9]

But what of the deterministic processes themselves? In Kahn's original work, they are treated as functions on channel histories without any internal structure. In a λ_{par} formulation of KPNs, they take the form of recursive functions that carry their state (and cursor positions) as arguments. In Figure 9, we use self-application to enable recursion, and we express a stream filter `filterDups` that prunes out all duplicate consecutive numbers from a stream.

This example assumes quite a bit in the way of syntactic sugar, though nothing non-standard. Church numerals would be needed to encode natural numbers, as well as a standard encoding of booleans. Because the above encoding will only work for finite executions, the cnt argument tells `filterDups` how many input elements to process. The fourth argument, lst, tracks the previously observed element on the input stream, this argument is the *state* of the stream transducer. The second and third arguments to `filterDups` are the cursors that track positions in both the input and output streams. The convert function is necessary for *computing* query sets based on the values of cursors.

This technique is sufficient for encoding arbitrary KPN programs into λ_{par}. The above encoding is by no means a natural expression of this concept, especially due to the fact that the input and output stream cursors must be tracked explicitly. However, with additional infrastructure for tracking stream cursors (and other

[9] In the λ_{par} abstraction we don't address concrete representations or storage requirements for LVar states and query sets. In a practical implementation, one would expect that already-consumed FIFO elements would be garbage collected, which in turn *requires* strict enforcement of consecutively increasing access only.

```
let filterDups = λf i1 i2 lst cnt.
    let next = get inp (convert i1)
      i2' = if (lst = next) then i2
            else put outp (convert (i2, next)); (i2 + 1)
    in if (cnt = 0) then {}
       else f f (i1 + 1) i2' next (cnt − 1)
in filterDups filterDups 0 ...
where convert i = {{(i, n)} | n ∈ ℕ}
      convert (i, n) = {{(i, n)}}
```

Figure 9. Process an input stream, removing consecutive duplicates. *inp* and *outp* are channels, globally bound elsewhere.

state) by means of a state monad, the above program could become significantly more idiomatic.

6. Building on the λ_{par} Foundation

The basic λ_{par} model, as we've presented it, is useful for unifying a body of work on existing programming models as well as suggesting some generalizations to real libraries like `monad-par` [18]. However, we believe that λ_{par} is also useful as a starting point for further extension and exploration, including limited forms of nondeterminism.

In this section we give an example of a natural extension of λ_{par} that provides additional capabilities. We use as our motivating example *asynchronous parallel reductions* [19], which are reductions (also known as *fold*s) that are disassociated from the parallel control flow of a program. Contributions to the reduction can come from any "thread" at any time, as can a demand for the final result.

By contrast, many parallel programming mechanisms provide *synchronous* reduction mechanisms, where the reduction is bound to the explicit (typically fork-join) parallel control flow. Often it is even known exactly how many parallel computations will participate in the reduction before the parallel region is entered (as in the Intel Threading Building Blocks library's `parallel_reduce` [21]). Typically the programmer may only: (1) initiate a parallel loop, (2) associate a reduction with that loop, (3) retrieve the value of the reduction only in the sequential region after the loop completes, *i.e.*, after the barrier. This mechanism is less flexible than asynchronous reductions.

6.1 Prelude: Syntactic Sugar for Counting

We will use a sum reduction, in which we fold the + operator over natural numbers, as our example. To do so, we will need to represent shared, increment-only counters as a λ_{par} domain like that shown in Figure 1(c). Doing so does not require an extension to λ_{par}, but it merits additional syntactic sugar in our meta-notation.

Rather than use the domain in Figure 1(c) directly, we simulate it using a power-set lattice (*i.e.*, ordered by subset inclusion), over an arbitrary alphabet of symbols: $\{a, b, c, \ldots\}$. LVars occupying such a lattice encode natural numbers using the cardinality of the subset.[10] Thus, a blocking get operation that unblocks when the count reaches, say, 3 would take a query set enumerating all the three-element subsets of the alphabet. We will use get l $|3|$ as a shorthand for this operation.

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

$$
\begin{aligned}
[\![\mathsf{unique}]\!] &= \lambda p.\ \mathsf{convert}\ p \\
[\![v]\!] &= \lambda p.\ v \\
[\![Q]\!] &= \lambda p.\ Q \\
[\![\lambda v.\ e]\!] &= \lambda p.\ \lambda v.\ [\![e]\!] \\
[\![\mathsf{new}]\!] &= \lambda p.\ new \\
[\![e1\ e2]\!] &= \lambda p.\ (([\![e1]\!]\ L{:}p)\ ([\![e2]\!]\ R{:}p)\ J{:}p) \\
[\![\mathsf{put}\ a\ b]\!] &= \lambda p.\ \mathsf{put}\ ([\![a]\!]\ L{:}p)\ ([\![b]\!]\ R{:}p) \\
[\![\mathsf{get}\ a\ b]\!] &= \lambda p.\ \mathsf{get}\ ([\![a]\!]\ L{:}p)\ ([\![b]\!]\ R{:}p) \\
[\![\mathsf{convert}\ e]\!] &= \lambda p.\ \mathsf{convert}\ ([\![e]\!]\ p)
\end{aligned}
$$

Figure 10. Rewrite rules for desugaring the unique construct within λ_{par} programs. Here we use "L:", "R:", "J:" to *cons* onto the front of a list that represents a path within a fork/join DAG. The symbols mean, respectively, "left branch", "right branch", or "after the join" of the two branches. This 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 domain D.

With this encoding, incrementing a shared variable l requires put l $\{\alpha\}$, where $\alpha \in \{a, b, c, \ldots\}$ and α has not previously been used. Thus, without any additional support, a hypothetical programmer would be responsible for creating a unique α for each parallel contribution to the counter.

There are well-known techniques, however, for generating a unique (but schedule-invariant and deterministic) identifier for a given point in a parallel execution. One solution 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 operation's *pedigree* and uses it to seed a deterministic parallel random number generator [17].

For brevity in our examples, we will assume a new syntactic sugar in the form of an expression unique, which, when evaluated, returns a singleton query set containing a single unique element of the alphabet: $\{\alpha\}$. With the unique syntax, we can write programs like the following, in which two parallel threads increment the same counter:

```
let sum = new in
  let par p1 = (put sum unique; put sum unique)
          p2 = (put sum unique)
  in ...
```
(Example 8)

In this case, the p_1 and p_2 "threads" will together increment the sum by three. Notice that consecutive increments performed by p_2 are not atomic. As a further shorthand, we will use $(\mathsf{bump}_1\ l)$ as synonymous with (put l unique), and $(\mathsf{bump}_n\ l)$ for n consecutive bump_1 expressions.

The unique construct could be implemented by a whole-program transformation over a sugared λ_{par} expression. Figure 10 shows one possible implementation. It simply creates a tree that tracks the dynamic evaluation of applications, and shows some similarity to a continuation-passing style transformation [9].

6.2 Extension: Destructive Observations with consume

With incrementable counters in place, we need only one more puzzle piece for asynchronous reductions: the ability to make *destructive reads* of the state of an LVar. We begin with the observation that when the state of a shared counter has come to rest—when no more increments will occur—then its final value is a deterministic function of program inputs, and is therefore safe to read directly. The problem is determining *when* an LVar has come to rest.

Here we use a speculative strategy: *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 increments will lead to an error. We can accomplish this by adding an extra state, called *probation*, to the domain D. The lattice defined by the relation $\sqsubseteq$ is extended thus:

$$\textit{probation} \sqsubseteq \top$$
$$\forall d \in D.\ d \not\sqsupseteq \textit{probation}$$

We then propose a new operation, consume, defined as follows:

- consume 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 (not a lower bound on that state).

The idea is to ensure that, after a consume, any further operations on l will go awry: get operations will block indefinitely, and put operations will attempt to move the state of l to $\top$, which will cause the system to step to **error**.

Below is an example program that uses consume to perform an asynchronous sum-reduction over a known number of inputs, but in an otherwise asynchronous environment. The (get cnt $|3|$) before the call to consume serves as a synchronization mechanism, ensuring that all increments are complete before the value is read.

$$\begin{array}{l}\textsf{let } cnt = \textsf{new in}\\ \quad \textsf{let } sum = \textsf{new in}\\ \quad\quad \textsf{let par } p_1 = (\textsf{bump}_3\ sum;\ \textsf{bump}_1\ cnt)\\ \quad\quad\quad\quad\ p_2 = (\textsf{bump}_4\ sum;\ \textsf{bump}_1\ cnt) \qquad \text{(Example 9)}\\ \quad\quad\quad\quad\ p_3 = (\textsf{bump}_5\ sum;\ \textsf{bump}_1\ cnt)\\ \quad\quad\quad\quad\ r\ \ = (\textsf{get } cnt\ |3|;\ \textsf{consume } sum)\\ \quad\quad \textsf{in convert } r\end{array}$$

This program executes three writers and one reader in parallel. Only when all writers complete does the reader return the sum, which in this case will be $3 + 4 + 5 = 12$. One may comment that the program could be simpler if we moved the consume after the final in, using the implicit barrier to synchronize the reduction. This is true; nevertheless, the desired example is one of asynchronous reduction where data dependencies alone suffice [19].

6.3 Limited Nondeterminism

The good news is that the above program is correct and *safe*; it will always return the same value in any execution. The bad news is that consume as a primitive operation *is not fully safe in general*. Rather, $\lambda_{\textsf{par}}$ + consume is an example of an intermediate class of languages exhibiting *limited* nondeterminism. A program in this language retains these properties:

- **(Property 1)** All runs of the program will terminate with the same value *if* they terminate without error.
- **(Property 2)** 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 the above program in which get cnt $|3|$ is replaced by get cnt $|2|$. 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. Nevertheless, there is still a benefit in the assurance that the program will never return a *wrong* non-**error** answer. Further, this class of languages can *still* shine compared to a conventional unsafe language when it comes to debugging, because when an error occurs it is possible to replay the same program with a tool that is guaranteed to reproduce exactly that error (without testing all possible thread interleavings).

As an example, we have implemented a *data-race detector*, along with a set of other $\lambda_{\textsf{par}}$ interpreters (`http://github.com/rrnewton/lambdapar_interps`). The algorithm uses the notion of pedigree in Figure 10, except that get and put are rewritten to include an additional p argument for pedigrees, *i.e.*, $\lambda p.$ get $([\![a]\!]\ L{:}p)$ $([\![b]\!]\ R{:}p)$ p. Next, the program is run with a modified implementation of get/put/consume that logs all effects to each location, along with their pedigree.[11] The LRJ pedigrees of Figure 10 already create a partial order, where, for example:

- $p < R{:}p$, *i.e.* p is *earlier* in time than its right child,
- $\forall q\ .\ q{:}R{:}p < J{:}p$, all pedigrees within a branch happen earlier than the join point.

Pedigrees *not* ordered by this partial order occur *in parallel*. Communication through LVars further constrains this time ordering (much like *vector clocks*). The log of effects is aggregated into a table of additional ordering constraints: *i.e.*, a put that unblocks a parallel get adds an edge. In this enhanced relation, if any consume operation is not strictly later in time than a put or get on the same location, then that consume is insufficiently synchronized.

7. Related Work

Work on deterministic parallel programming models is long-standing. In addition to the single-assignment and data-flow languages already discussed, here we consider a few more 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 $\lambda_{\textsf{par}}$'s multiple monotonic assignment would likely involve parameterizing the type system by the user-specified partial order—a direction of inquiry that we leave for future work.

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 $\lambda_{\textsf{par}}$ model and DPJ is that $\lambda_{\textsf{par}}$ retains determinism by restricting *what* can be read or written, rather than by restricting *who* can read or write.

Concurrent Revisions The Concurrent Revisions (CR) [16] programming model uses isolation types to distinguish regions of the heap shared by multiple mutators. Rather than enforcing exclusive access, CR clones a fresh copy of the state for each mutator, using a deterministic policy for resolving conflicts in these 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* [12]). CR charts an important new area in the deterministic-parallelism design space, but one that differs significantly from $\lambda_{\textsf{par}}$. 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 visi-

[11] Unfortunately, while there are significant results reducing the space cost of data-race detection for strictly nested, fork-join parallel models [5], for the more general class of dependence graphs induced by $\lambda_{\textsf{par}}$, there are not yet known methods for reducing the space requirements for data-race detection to less than the space required to store the entire dynamic dependence graph itself.

ble at the end of parallel regions, rather than λ_{par}'s asynchronous communication within parallel regions.

Bloom and Bloom^L In the database literature, there is a body of work on *monotonic logic* which has recently been leveraged by the Bloom language [1] to achieve *eventual consistency* in distributed programs. Bloom programs describe relationships between distributed data collections that are updated monotonically. The initial formulation of Bloom [2] had a notion of monotonicity based on *set containment*, corresponding to the store ordering for single-assignment languages given in Definition 4. However, a very recent technical report by Conway *et al.* [8] generalizes Bloom to a lattice-parameterized system, Bloom^L, that comes with a library of built-in lattice types and also allows for users to implement their own lattice types as Ruby classes. While Conway *et al.* do not give a proof of eventual consistency for Bloom^L, our determinism result for λ_{par} suggests that generalizing from a set-containment-based (in our setting, IVar-based) notion of monotonicity to one parameterized by a user-defined lattice (LVar-based) is indeed a safe transformation. Moreover, although Bloom is very different from λ_{par}, we believe that Bloom bodes well for programmers' willingness to use lattice-based data types like LVars, and lattice-parameterizable languages like λ_{par}, to address practical programming challenges.

Quantum programming The λ_{par} semantics is reminiscent of the semantics of quantum programming languages that extend a conventional λ-calculus with a store that maintains the quantum state. Because of quantum parallelism, the quantum state can be accessed by many threads in parallel, but only through a restricted interface. As a concrete example, the language designed by Selinger and Valiron [22] allows only the following operations on quantum data: (1) "appending" to the current data using the tensor product; (2) performing a unitary operation that must, by definition, act linearly and uniformly on the data; and (3) selecting a set of orthogonal subspaces and performing a measurement that projects the quantum state onto one of the subspaces. These operations correspond roughly to λ_{par}'s new, put, and get. Quantum mechanics may serve as a source of inspiration when designing operations like consume that introduce limited nondeterminism.

8. Conclusion

We have presented and formally proved determinism for λ_{par}, a shared-state parallel programming model that generalizes and unifies existing single-assignment and data-flow models and provides a foundation for exploration of limited nondeterminism. Future work will formally prove the more limited guarantees provided by $\lambda_{\mathsf{par}} + \mathsf{consume}$, and will explore formal methods for verifying the safety of `monad-par` data-structure libraries against a λ_{par} model.

Acknowledgments

We thank Amr Sabry for his feedback and tireless assistance through all stages of this work.

References

[1] URL `http://bloom-lang.net`.

[2] P. Alvaro, N. Conway, J. Hellerstein, and W. R. Marczak. Consistency analysis in Bloom: a CALM and collected approach. In *CIDR*, pages 249–260. www.cidrdb.org, 2011.

[3] Arvind, R. S. Nikhil, and K. K. Pingali. I-structures: data structures for parallel computing. *ACM Trans. Program. Lang. Syst.*, 11:598–632, October 1989. ISSN 0164-0925.

[4] F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.

[5] M. A. Bender, J. T. Fineman, S. Gilbert, and C. E. Leiserson. On-the-fly maintenance of series-parallel relationships in fork-join multithreaded programs. In *Proc. of the Symposium on Parallelism in Algorithms and Architectures*, SPAA '04, New York, NY, USA, 2004.

[6] R. L. Bocchino, Jr. et al. Safe nondeterminism in a deterministic-by-default parallel language. In *Principles of Programming Languages*, POPL '11, pages 535–548, New York, NY, USA, 2011.

[7] Z. Budimlić, M. Burke, V. Cavé, K. Knobe, G. Lowney, R. Newton, J. Palsberg, D. Peixotto, V. Sarkar, F. Schlimbach, and S. Taşirlar. Concurrent collections. *Sci. Program.*, 18:203–217, August 2010. ISSN 1058-9244. URL `http://dl.acm.org/citation.cfm?id=1938482.1938486`.

[8] N. Conway, W. Marczak, P. Alvaro, J. M. Hellerstein, and D. Maier. Logic and lattices for distributed programming. Technical Report UCB/EECS-2012-167, EECS Department, University of California, Berkeley, Jun 2012.

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

[10] M. Felleisen, R. B. Findler, and M. Flatt. *Semantics Engineering with PLT Redex*. The MIT Press, 1st edition, 2009. ISBN 0262062755, 9780262062756.

[11] M. Fluet, M. Rainey, J. Reppy, and A. Shaw. Implicitly-threaded parallelism in manticore. In *Proc. of the International Conference on Functional Programming*, ICFP '08, New York, NY, USA, 2008.

[12] M. Frigo et al. Reducers and other cilk++ hyperobjects. In *Symposium on Parallelism in Algorithms and Architectures*, SPAA '09, New York, NY, USA.

[13] J. Hicks, D. Chiou, B. S. Ang, and Arvind. Performance studies of id on the monsoon dataflow system. *J. Parallel Distrib. Comput.*, 1993.

[14] G. Kahn. The semantics of a simple language for parallel programming. In J. L. Rosenfeld, editor, *Information processing*, pages 471–475, Stockholm, Sweden, Aug 1974. North Holland, Amsterdam.

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

[16] D. Leijen, M. Fahndrich, and S. Burckhardt. Prettier concurrency: purely functional concurrent revisions. In *Proc. of the ACM Haskell Symposium*, Haskell '11, 2011.

[17] C. E. Leiserson, T. B. Schardl, and J. Sukha. Deterministic parallel random-number generation for dynamic-multithreading platforms. In *Principles and Practice of Parallel Programming*, PPoPP '12, 2012.

[18] S. Marlow, R. Newton, and S. Peyton Jones. A monad for deterministic parallelism. In *Proceedings of the 4th ACM symposium on Haskell*, Haskell '11, New York, NY, USA, 2011. ACM.

[19] R. Newton et al. Deterministic reductions in an asynchronous parallel language. In *Workshop on Determinism and Correctness in Parallel Programming (WoDet)*, Newport Beach, CA, 2011.

[20] P. W. O'Hearn, J. C. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In L. Fribourg, editor, *CSL*, volume 2142 of *Lecture Notes in Computer Science*, pages 1–19. Springer, 2001. ISBN 3-540-42554-3.

[21] J. Reinders. *Intel Threading Building Blocks*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, first edition, 2007. ISBN 9780596514808.

[22] P. Selinger and B. Valiron. A lambda calculus for quantum computation with classical control. *Mathematical Structures in Computer Science*, 16(3):527–552, 2006.

[23] D. Spoonhower, G. E. Blelloch, P. B. Gibbons, and R. Harper. Beyond nested parallelism: tight bounds on work-stealing overheads for parallel futures. In *Proc. of the Symposium on Parallelism in Algorithms and Architectures*, SPAA '09, New York, NY, USA, 2009. ACM.

[24] L. G. Tesler and H. J. Enea. A language design for concurrent processes. In *Proc. of the April 30–May 2, 1968, spring joint computer conference*, AFIPS '68 (Spring), pages 403–408, New York, NY, USA, 1968.