LATTICE-BASED DATA STRUCTURES FOR DETERMINISTIC PARALLEL AND DISTRIBUTED PROGRAMMING

Lindsey Kuper

Submitted to the faculty of the University Graduate School
in partial fulfillment of the requirements
for the degree
Doctor of Philosophy
in the School of Informatics and Computing
Indiana University
September 2014

Accepted by the Graduate Faculty, Indiana University, in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Ryan R. Newton, Ph.D.

Lawrence S. Moss, Ph.D.

Amr Sabry, Ph.D.

Chung-chieh Shan, Ph.D.

09/08/2014

Acknowledgements

TODO: Write acks.

For Alex, who said,
“You’re gonna destroy grad school.”

LATTICE-BASED DATA STRUCTURES FOR DETERMINISTIC PARALLEL AND DISTRIBUTED PROGRAMMING

Lindsey Kuper

Deterministic-by-construction parallel programming models guarantee that programs have the same observable behavior on every run, promising freedom from bugs caused by schedule nondeterminism. To make that guarantee, though, they must sharply restrict sharing of state between parallel tasks, usually either by disallowing sharing entirely or by restricting it to one type of data structure, such as single-assignment locations.

I show that *lattice-based* data structures, or *LVars*, are the foundation for a guaranteed-deterministic parallel programming model that allows a more general form of sharing. LVars allow multiple assignments that are inflationary with respect to an application-specific lattice. They ensure determinism by allowing only inflationary writes and "threshold" reads that block until a lower bound is reached. After presenting the basic LVars model, I extend it to support *event handlers*, which enable an event-driven programming style, and non-blocking "freezing" reads, resulting in a *quasi-deterministic* model in which programs behave deterministically modulo exceptions.

I demonstrate the viability of the LVars model with *LVish*, a Haskell library that provides a collection of lattice-based data structures, a work-stealing scheduler, and a monad in which LVar computations run. LVish leverages Haskell's type system to index such computations with *effect levels* to ensure that only certain LVar effects can occur, hence statically enforcing determinism or quasi-determinism. I present two case studies of parallelizing existing programs using LVish: a k-CFA control flow analysis, and a bioinformatics application for comparing phylogenetic trees.

Finally, I show how LVar-style threshold reads apply to the setting of *convergent replicated data types* (CvRDTs), which specify the behavior of eventually consistent replicated objects in a distributed system.

I extend the CvRDT model to support deterministic, strongly consistent *threshold queries*. The technique generalizes to any lattice, and hence any CvRDT, and allows deterministic observations to be made of replicated objects before the replicas' states converge.

Contents

CHAPTER 1

Introduction

Parallel programming—that is, writing programs that can take advantage of parallel hardware to go faster—is notoriously difficult. A fundamental reason for this difficulty is that programs can yield inconsistent results, or even crash, due to unpredictable interactions between parallel tasks.

Deterministic-by-construction parallel programming models, though, offer the promise of freedom from subtle, hard-to-reproduce nondeterministic bugs in parallel code. While there are many ways to construct deterministic parallel programs and to verify the determinism of individual programs, only a deterministic-by-construction programming model can provide a *language-level* guarantee of determinism: deterministic programs are the only programs that can be expressed within the model.

A deterministic-by-construction programming model is one that ensures that all programs written using the model have the same *observable behavior* every time they are run. How do we define what is observable about a program's behavior? Certainly, we do *not* wish to preserve behaviors such as running time across multiple runs—ideally, a deterministic-by-construction parallel program will run faster when more parallel resources are available. Moreover, we do not want to count scheduling behavior as observable—in fact, we want to specifically *allow* tasks to be scheduled dynamically and unpredictably, without allowing such *schedule nondeterminism* to affect the observable behavior of a program. Therefore, in this dissertation I will define the observable behavior of a program to be *the value to which the program evaluates*.

LK: In my proposal, I had a footnote here: "We assume that programs have no side effects other than state effects." I think I instead just want to say that we *ignore* other side effects. They can *happen*; it's just that they don't count.

This definition of observable behavior ignores side effects other than *state*. Even so, sharing of state between parallel computations raises the possibility of *race conditions* that allow schedule nondeterminism to be observed in the outcome of a program. For instance, if a computation writes 3 to a shared location while another computation writes 4, then a subsequent third computation that reads and returns the location's contents will nondeterministically return 3 or 4, depending on the order in which the first two computations ran. Therefore, if a parallel programming model is to guarantee determinism by construction, it must necessarily limit sharing of mutable state between parallel tasks in some way.

1.1. The deterministic-by-construction parallel programming landscape

There is long-standing work on deterministic-by-construction parallel programming models that limit sharing of state between tasks. The possibilities include:

- *No-shared-state parallelism.* One classic approach to guaranteeing determinism in a parallel programming model is to allow *no* shared mutable state between tasks, forcing tasks to produce values independently. An example of no-shared-state parallelism is pure functional programming with function-level task parallelism, or *futures*—for instance, in Haskell programs that use the `par` and `pseq` combinators [34]. The key characteristic of this style of programming is lack of side effects: because programs don't have side effects, expressions can evaluate simultaneously without affecting the eventual value of the program. Also belonging in this category are parallel programming models based on *pure data parallelism*, such as Data Parallel Haskell [41, 14] or the River Trail API for JavaScript [27], each of which extend existing languages with *parallel array* data types and (observably) pure operations on them. LK: Does it make sense to say that DPH is observably pure? It does mutate arrays.
- *Data-flow parallelism.* In *Kahn process networks* (KPNs) [29], as well as in the more restricted *synchronous data flow* systems [31], a network of independent "computing stations" communicate with each other through first-in first-out (FIFO) queues, or *channels*. Reading data out of such a FIFO queue is a *blocking* operation: once an attempt to read has started, a computing station cannot do anything else until the data to be read is available. Each station computes a sequential, monotonic function from the *history* of its input channels (*i.e.*, the input it has received so far) to the history

of its output channels (the output it has produced so far). KPNs are the basis for deterministic stream-processing languages such as StreamIt [26].

- *Single-assignment parallelism.* In parallel *single-assignment* languages, "full/empty" bits are associated with memory locations so that they may be written to at most once. Single-assignment locations with blocking read semantics are known as *IVars* [3] and are a well-established mechanism for enforcing determinism in parallel settings: they have appeared in Concurrent ML as `SyncVars` [43]; in the Intel Concurrent Collections (abbreviated "CnC") system [11]; 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 [35] uses IVars in a deterministic-by-construction setting, allowing user-created threads to communicate through IVars without requiring the `IO` monad. Rather, operations that read and write IVars must run inside a `Par` monad, thus encapsulating them inside otherwise pure programs, and hence a program in which the only effects are `Par` effects is guaranteed to be deterministic.
- *Imperative disjoint parallelism.* Finally, yet another approach to guaranteeing determinism is to ensure that the state accessed by concurrent threads is *disjoint*.LK: This is the first place I've used the word "concurrent". Should I explain concurrency vs. parallelism in a footnote or something? Is it even a useful distinction in this context? Sophisticated permissions systems and type systems make it possible for imperative programs to mutate state in parallel, while guaranteeing that the same state is not accessed simultaneously by multiple threads. I will refer to this style of programming as *imperative disjoint parallelism*, with Deterministic Parallel Java (DPJ) [8, 7] as a prominent example.

The four parallel programming models listed above—no-shared-state parallelism, data-flow parallelism, single-assignment parallelism, and imperative disjoint parallelism—all seem to embody rather different mechanisms for exposing parallelism and for ensuring determinism. If we view these different programming models as a toolkit of unrelated choices, though, it is not clear how to proceed when we want to implement an application with multiple parallelizable components that are best suited to different programming models. For example, suppose we have an application in which we wish to use data-flow pipeline parallelism via FIFO queues, but also disjoint parallel mutation of arrays. It is not

obvious how to compose two programming models that each only allow communication through a single type of shared data structure—or, if we do manage to compose them, whether or not the determinism guarantee of the individual models is preserved by their composition. Therefore, we seek a general, broadly-applicable model for deterministic parallel programming that is not tied to a particular data structure.

1.2. Lattice-based, monotonic data structures as a basis for deterministic parallelism

In KPNs and other data-flow models, communication takes place over blocking FIFO queues with ever-increasing *channel histories*, while in IVar-based programming models such as CnC and monad-par, a shared data store of blocking single-assignment memory locations grows monotonically. Hence *monotonic data structures*—data structures to which information can only be added and never removed—emerge as a common theme of guaranteed-deterministic programming models.[1]

In this dissertation, I show that *lattice-based* data structures, or *LVars*, are the foundation for a model of deterministic-by-construction parallel programming that allows a more general form of communication between tasks than previously existing guaranteed-deterministic models allowed. LVars generalize IVars and are so named because the states an LVar can take on are elements of an application-specific *lattice*.[2] This application-specific lattice determines the semantics of the `put` and `get` operations that comprise the interface to LVars (which I will explain in detail in Chapter 2):

- The `put` operation can only change an LVar's state in a way that is *monotonically increasing* with respect to the 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.

[1] In Chapter 2, I will refine this definition of "monotonic data structures" to mean data structures to which information can only be added and never removed, *and* for which the order in which information is added is not observable.

[2] As I will explain in Chapter 2, what I am calling a "lattice" here really need only be a *bounded join-semilattice* augmented with a greatest element $\top$.

Together, least-upper-bound writes via `put` and threshold reads via `get` yield a deterministic-by-construction programming model. That is, a program in which `put` and `get` operations on LVars are the only side effects will have the same observable result in spite of parallel execution and schedule nondeterminism. LK: Maybe I don't need the next sentence at all; maybe it's covered by the "organization" bullet points below. As we will see in Chapter 2, no-shared-state parallelism, data-flow parallelism and single-assignment parallelism are all subsumed by the LVars programming model, and as we will see in Section 4.3, imperative disjoint parallel updates are compatible with LVars as well. Furthermore, as I show in Section 2.6, we can generalize the behavior of the `put` and `get` operations while retaining determinism: we can generalize `put` from least-upper-bound writes to non-idempotent (but still inflationary and commutative) writes, and we can generalize the `get` operation to allow a more general form of threshold reads.

1.3. Quasi-deterministic and event-driven programming with LVars

The LVars model described above guarantees determinism and supports an unlimited variety of shared data structures: anything viewable as a lattice. However, it is not as general-purpose as one might hope. Consider, for instance, an algorithm for unordered graph traversal. A typical implementation involves a monotonically growing set of "seen nodes"; neighbors of seen nodes are fed back into the set until it reaches a fixed point. Such fixpoint computations are ubiquitous, and would seem to be a perfect match for the LVars model due to their use of monotonicity. But they are not expressible using the threshold `get` and least-upper-bound `put` operations, nor even with the more general alternatives to `get` and `put` mentioned above.

The problem is that these computations rely on *negative* information about a monotonic data structure, *i.e.*, on the *absence* of certain writes to the data structure. In a graph traversal, for example, neighboring nodes should only be explored if the current node is *not yet* in the set; a fixpoint is reached only if no new neighbors are found; and, of course, at the end of the computation it must be possible to learn exactly which nodes were reachable (which entails learning that certain nodes were not). In Chapter 3, I describe two extensions to the basic LVars model that make such computations possible:

- First, I extend the model with a primitive operation `freeze` for *freezing* an LVar, which allows its contents to be read immediately and exactly, rather than the blocking threshold read that `get` allows. The `freeze` primitive imposes the following trade-off: once an LVar has been frozen, any further writes that would change its value instead raise an exception; on the other hand, it becomes possible to discover the exact value of the LVar, learning both positive and negative information about it, without blocking. Therefore, LVar programs that use `freeze` are *not* guaranteed to be deterministic, because they could nondeterministically raise an exception depending on how `put` and `freeze` operations are scheduled. However, such programs satisfy *quasi-determinism*: all executions that produce a final value produce the *same* final value.
- Second, I add the ability to attach *event handlers* to an LVar. When an event handler has been registered with an LVar, it invokes a *callback function* to run, asynchronously, whenever events arrive (in the form of monotonic updates to the LVar). Crucially, it is possible to check for *quiescence* of a group of handlers, discovering that no callbacks are currently enabled—a transient, negative property. Since quiescence means that there are no further changes to respond to, it can be used to tell that a fixpoint has been reached.

Of course, since more events could arrive later, there is no way to guarantee that quiescence is permanent—but since the contents of the LVar being written to can only be read through `get` or `freeze` operations anyway, early quiescence poses no risk to determinism or quasi-determinism, respectively. In fact, freezing and quiescence work particularly well together because freezing provides a mechanism by which the programmer can safely "place a bet" that all writes have completed. Hence freezing and handlers make possible fixpoint computations like the graph traversal described above. Moreover, if we can ensure that the freeze does indeed happen after all writes have completed, then we can ensure that the computation is deterministic, and it is possible to enforce this "freeze-last" idiom at the implementation level, as I discuss below (and, in more detail, in Section 4.2.6).

1.4. The LVish library

To demonstrate the practicality of the LVars programming model, in Chapter 4 I will describe *LVish*,[3] a Haskell library for deterministic and quasi-deterministic programming with LVars.

LVish provides a `Par` monad for encapsulating parallel computation and enables a notion of lightweight, library-level threads to be employed with a custom work-stealing scheduler.[4] LVar computations run inside the `Par` monad, which is indexed by an *effect level*, allowing fine-grained specification of the effects that a given computation is allowed to perform. For instance, since `freeze` introduces quasi-determinism, a computation indexed with a deterministic effect level is not allowed to use `freeze`. Thus, the *type* of an LVish computation reflects its determinism or quasi-determinism guarantee. Furthermore, if a `freeze` is guaranteed to be the *last* effect that occurs in a computation, then it is impossible for that `freeze` to race with a `put`, ruling out the possibility of a run-time `put`-after-`freeze` exception. LVish exposes a `runParThenFreeze` operation that captures this "freeze-last" idiom and has a deterministic effect level.

LVish also provides a variety of lattice-based data structures (*e.g.*, sets, maps, graphs) that support concurrent insertion, but not deletion, during `Par` computations. In addition to those that LVish provides, users may implement their own lattice-based data structures, and LVish provides tools to facilitate the definition of user-defined LVars. I will describe the proof obligations for data structure implementors and give examples of applications that use user-defined LVars as well as those that the library provides.

In addition to discussing the implementation of the LVish library and introducing the above features with examples, Chapter 4 illustrates LVish through two case studies, drawn from my collaborators' and my experience using the LVish library, both of which make use of handlers and freezing:

- First, I describe using LVish to parallelize a control flow analysis (k-CFA) algorithm. The goal of k-CFA is to compute the flow of values to expressions in a program. The k-CFA algorithm proceeds in two phases: first, it explores a graph of *abstract states* of the program; then, it summarizes the

[3] Available at `http://hackage.haskell.org/package/lvish`.

[4] The `Par` monad exposed by LVish generalizes the original `Par` monad exposed by the *monad-par* library (`http://hackage.haskell.org/package/monad-par`, described by Marlow *et al.* [35]), which allows determinism-preserving communication between threads, but only through IVars, rather than LVars.

results of the first phase. Using LVish, these two phases can be pipelined in a manner similar to the pipelined breadth-first graph traversal described above; moreover, the original graph exploration phase can be internally parallelized. I contrast the LVish implementation with the original sequential implementation and give performance results.

- Second, I describe using LVish to parallelize *PhyBin* [39], a bioinformatics application for comparing sets of phylogenetic trees that relies heavily on a parallel tree-edit distance algorithm [48]. In addition to handlers and freezing, the PhyBin application crucially relies on the aforementioned ability to perform non-idempotent (but inflationary and commutative) writes to LVars (in contrast to least-upper-bound writes discussed above, which are idempotent as well as inflationary and commutative).

1.5. Deterministic threshold queries of distributed data structures

The LVars model is closely related to the concept of *conflict-free replicated data types* (CRDTs) [45] for enforcing *eventual consistency* [52] of replicated objects in a distributed system. In particular, *state-based* or *convergent* replicated data types, abbreviated as *CvRDTs* [45, 44], leverage the mathematical properties of join-semilattices to guarantee that all replicas of an object (for instance, in a distributed database) eventually agree.

Although CvRDTs are provably eventually consistent, queries of CvRDTs (unlike threshold reads of LVars) nevertheless allow inconsistent intermediate states of replicas to be observed. That is, if two replicas of a CvRDT object are updated independently, reads of those replicas may disagree until a (least-upper-bound) *merge* operation takes place.

Taking inspiration from LVar-style threshold reads, in Chapter 5 I show how to extend CvRDTs to support deterministic, *strongly consistent* queries using a mechanism called *threshold queries* (or, seen from another angle, I show how to port threshold reads from a shared-memory setting to a distributed one). The threshold query technique generalizes to any lattice, and hence any CvRDT, and allows deterministic observations to be made of replicated objects before the replicas' states have converged. This work has practical relevance since, while real distributed database applications call for a combination of eventually consistent and strongly consistent queries, CvRDTs only support the former, and threshold queries

extend the CvRDT model to support both kinds of queries within a single, lattice-based reasoning framework. Furthermore, since threshold queries behave deterministically regardless of whether all replicas agree, they suggest a way to save on synchronization costs: existing operations that require all replicas to agree could be done with threshold queries instead, and retain behavior that is *observably* strongly consistent while avoiding unnecessary synchronization.

1.6. Thesis statement, and organization of the rest of this dissertation

With the above background, I can state my thesis:LK: This format ripped off from Josh Dunfield.

> Lattice-based data structures are a general and practical foundation for deterministic and quasi-deterministic parallel and distributed programming.

The rest of this dissertation supports my thesis as follows:

- *Lattice-based data structures*: In Chapter 2, I formally define LVars and use them to define λ_{LVar}, a call-by-value parallel calculus with a store of LVars that support least-upper-bound `put` and threshold `get` operations. In Chapter 3, I extend λ_{LVar} to add support for event handlers and the `freeze` operation, calling the resulting language λ_{LVish}. Appendix B contains runnable versions of λ_{LVar} and λ_{LVish} implemented using the PLT Redex semantics engineering system [**20**] for interactive experimentation.
- *general*: In Chapter 2, I show how previously existing deterministic parallel programming models (single-assignment languages, Kahn process networks) are subsumed by the lattice-generic LVars model. Additionally, I show how to generalize the `put` and `get` operations on LVars while preserving their determinism.
- *deterministic*: In Chapter 2, I show that the basic LVars model guarantees determinism by giving a proof of determinism for the λ_{LVar} language with `put` and `get`.
- *quasi-deterministic*: In Chapter 3, I define quasi-determinism and give a proof of quasi-determinism for λ_{LVish}, which adds the `freeze` operation and event handlers to the λ_{LVar} language of Chapter 2. λ_{LVish} also generalizes the language from allowing only least-upper-bound writes to allowing arbitrary inflationary and commutative writes.

- *practical*: In Chapter 4, I describe the LVish Haskell library, which is based on the LVars programming model, and demonstrate how it is used for practical programming with the two case studies described above, including performance results.
- *distributed programming*: In Chapter 5, I show how LVar-style threshold reads apply to the setting of distributed, replicated data structures. In particular, I extend convergent replicated data types (CvRDTs) to support strongly consistent threshold queries, which take advantage of the existing lattice structure of CvRDTs and allow deterministic observations to be made of their contents without requiring all replicas to agree.

1.7. Previously published work

This dissertation draws heavily on the earlier work and writing appearing in the following papers, written jointly with several collaborators:LK: This exact wording stolen from Aaron Turon. maybe tweak it?

- Lindsey Kuper and Ryan R. Newton. 2013. LVars: lattice-based data structures for deterministic parallelism. In *Proceedings of the 2nd ACM SIGPLAN Workshop on Functional High-Performance Computing* (FHPC '13).
- Lindsey Kuper, Aaron Turon, Neelakantan R. Krishnaswami, and Ryan R. Newton. 2014. Freeze after writing: quasi-deterministic parallel programming with LVars. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (POPL '14).
- Lindsey Kuper, Aaron Todd, Sam Tobin-Hochstadt, and Ryan R. Newton. 2014. Taming the parallel effect zoo: extensible deterministic parallelism with LVish. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI '14).

LK: I was going to put something like, "The material in this chapter is based on research done jointly with..." in individual chapters, and cite each paper at the start of its chapter. But chapters don't really correspond to the papers, so I'm just going to try citing all the papers up front like this.

CHAPTER 2

LVars: lattice-based data structures for deterministic parallelism

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. While a number of popular languages and language extensions (*e.g.*, Cilk [24]LK: Any others?) *encourage* deterministic parallel programming, few of them guarantee determinism at the language level—that is, for *all* programs that can be written using the model.

Of the options available for parallel programming with a language-level determinism guarantee, perhaps the most mature and broadly available choice 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 [34].[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 the future is not yet available, the consumer immediately switches roles to begin computing the value (as explored by Marlow *et al*. [35]).

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 race conditions that introduce nondeterminism, so any parallel programming model that hopes to guarantee determinism must do something to tame sharing—that is, to restrict access to mutable state shared among concurrent computations. Systems such as Deterministic Parallel Java [8, 7], for instance,

[1] When programming with `par` and `pseq`, a language-level determinism guarantee obtains if user programs are written in the *Safe Haskell* [49] subset of Haskell (which is implemented in GHC Haskell by means of the `SafeHaskell` language pragma), and if they do not use the `IO` monad.

accomplish this by ensuring that the state accessed by concurrent threads is *disjoint*. Alternatively, a programming model might allow *data* to be shared, but limit the *operations* that can be performed on it to only those operations that commute with one another and thus can tolerate nondeterministic thread interleavings. In such a setting, although the order in which side-effecting operations occur can differ on multiple runs, a program will always produce the same observable result.[2]

In Kahn process networks (KPNs) [29] and other *data-flow parallel* models—which are the basis for deterministic stream-processing languages such as StreamIt [26]—communication among processes takes place over blocking FIFO queues with ever-increasing *channel histories*. Meanwhile, in *single-assignment* [51] or *IVar-based* [3] programming models, such as the Intel Concurrent Collections system (CnC) [11] and the *monad-par* Haskell library [35], a shared data store of blocking single-assignment memory locations grows monotonically. Hence *monotonic data structures*—data structures to which information can only be added and never removed, and for which the order in which information is added is not observable—emerge as a common theme of both data-flow and single-assignment parallel programming 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 systems like StreamIt, CnC, and monad-par emerge independently, without recognition of their common basis. Moreover, since each one of these programming models is based on a single type of shared data structure, they lack *generality*: IVars or FIFO streams alone cannot support all producer/consumer applications, as I discuss in Section 2.1.

Instead of limiting ourselves to a single type of shared data structure, though, we can take the more general notion of monotonic data structures as the basis for a new deterministic parallel programming model. In this chapter, I show how to generalize IVars to *LVars*, thus named because the states an LVar can take on are elements of an application-specific *lattice*.[3] This application-specific lattice determines

[2]There are many ways to define what is observable about a program. As noted in Chapter 1, I define the observable behavior of a program to be the value to which it evaluates.

[3]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; see Section 2.3.1. For brevity, I use the term "lattice" in place of "bounded join-semilattice with a designated greatest element" throughout this dissertation.

the semantics of the `put` and `get` operations that comprise the interface to LVars (which I will explain in detail in the sections that follow): the `put` operation takes the least upper bound of the current state and the new state with respect to the lattice, and the `get` operation performs a *threshold read* that blocks until a lower bound in the lattice is reached.

Section 2.2 introduces the concept of LVars through a series of examples. Then, in Sections 2.3 and 2.4 I formally 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 one "empty" state and multiple "full" states (where $\forall i.\ \mathit{empty} < \mathit{full}_i$) 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 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.

The main technical result of this chapter is a proof of determinism for λ_{LVar} (Section 2.5). The key lemma, Independence (Section 2.5.5), gives a kind of *frame property* for LVar computations: very roughly, if a computation takes an LVar from state d to d', then it would take the same LVar from the state $d \sqcup d_F$ to $d' \sqcup d_F$. The Independence lemma captures the commutative effects of LVar computations. Interestingly, such a property would *not* hold in a typical language with shared mutable state, but holds in the setting of λ_{LVar} because of the monotonic semantics of LVars and their `put`/`get` interface.

Finally, in Section 2.6, I consider some alternative semantics for the `put` and `get` operations that generalize their behavior while retaining the determinism of the original semantics: I generalize the `put` operation from least-upper-bound writes to inflationary, commutative writes, and I generalize the `get` operation to allow a more general form of threshold reads.

2.1. 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 mutable data structures that change monotonically. 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, consider a specific problem, drawn from the domain of *graph algorithms*, where issues of ordering create a tension between parallelism and determinism:

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

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

This is a challenge problem for deterministic-by-construction parallel programming: existing parallel solutions (such as, for instance, the parallel breadth-first graph traversal implementation from the Parallel Boost Graph Library [1]) often traverse the connected component in a nondeterministic order (even though the outcome of the program—that is, 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. Before turning to an LVar-based approach, though, let's consider whether a purely functional (and therefore deterministic by construction) program can meet the specification.

```
nbrs :: Graph → NodeLabel → Set NodeLabel
-- `nbrs g n' is the neighbor nodes of node `n' in graph `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 2.1. 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.

2.1.1. A purely functional attempt. Figure 2.1 gives a Haskell implementation of a *level-synchronized* breadth-first graph traversal that finds the connected component reachable from a starting vertex. 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 (in fact, the aforementioned implementation from the Parallel Boost Graph Library [1] uses level-synchronization), although it necessarily sacrifices some parallelism for determinism: parallel tasks cannot continue discovering nodes in the component before synchronizing with all other tasks at a given distance from the start.

Unfortunately, the code given in Figure 2.1 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 doing so just passes the 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 since it instead returns a *set*, 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.[4]

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, at the granularity of the individual level-sets, the problem would remain: within each level-set, one would not be able to launch all instances of `analyze` and asynchronously use those results that finished first. Moreover, we would still have to contend with the previously-mentioned difficulty of separating producers and consumers when expressing producer-consumer computations when using pure programming with futures [35].

2.1.2. An LVar-based solution. Consider a version of `bf_traverse` written using a programming model with limited effects that allows *any* data structure to be shared among tasks, 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, enabling pipelining, but may only observe irrevocable, monotonic properties of it. This is possible with a programming model based on LVars. In the rest of this chapter, I

[4] 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.

will formally introduce LVars and the λ_{LVar} language and give a proof of determinism for λ_{LVar}. Then, in Chapter 3, I will extend the basic LVars model with additional features that make it easier to implement parallel graph traversal algorithms and other fixpoint computations, and I will return to `bf_traverse` and show how to implement a version of it using LVars that makes pipelining possible and truly satisfies the above specification.

2.2. LVars by example

IVars [3, 40, 11, 35] are a well-known mechanism for deterministic parallel programming. An IVar is a *single-assignment* variable [51] with a blocking read semantics: an attempt to read an empty IVar will block until the IVar has been filled with a value. LVars are a generalization of IVars: unlike IVars, which can only be written to once, LVars allow multiple writes, so long as those writes are monotonically increasing with respect to an application-specific lattice of states.

Consider a program (written in a hypothetical language) in which two parallel computations write to an LVar *lv*, with one thread writing the value 2 and the other writing 3:

(Example 2.1)
$$\begin{aligned} &\texttt{let par } \_ = \texttt{put } \mathit{lv}\ 3 \\ &\qquad\quad \_ = \texttt{put } \mathit{lv}\ 2 \\ &\texttt{in get } \mathit{lv} \end{aligned}$$

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

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

has *fork-join* semantics: it launches concurrent subcomputations $e_1, e_2, \ldots$ whose executions arbitrarily interleave, but must all complete before *body* runs. The `put` operation is defined in terms of the application-specific lattice of LVar states: it updates the LVar to the *least upper bound* of its current state and the new state being written.

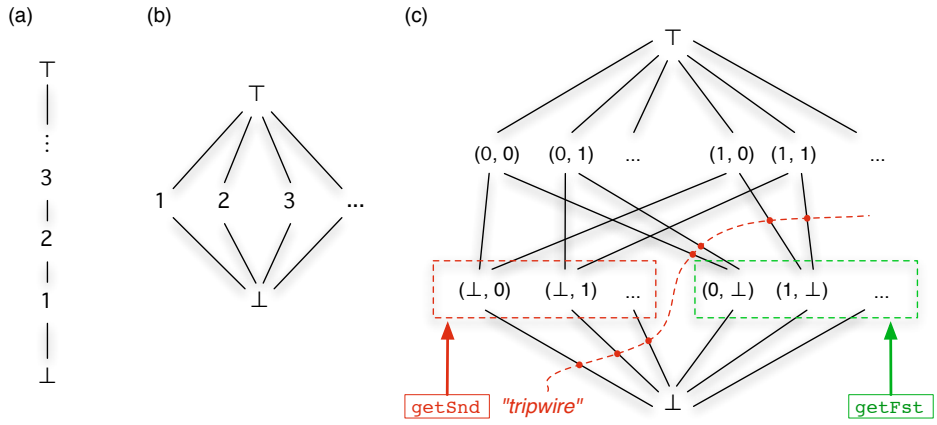

Figure 2.2. Example LVar lattices: (a) positive integers ordered by $\leq$; (b) IVar containing a positive integer; (c) pair of natural-number-valued IVars, annotated with example threshold sets that would correspond to a blocking read of the first or second element of the pair. Any state transition crossing the "tripwire" for `getSnd` causes it to unblock and return a result.

If *lv*'s lattice is the $\leq$ ordering on positive integers, as shown in Figure 2.2(a), then *lv*'s state will always be $\max(3, 2) = 3$ by the time `get` *lv* runs, since the least upper bound of two positive integers n_1 and n_2 is $\max(n_1, n_2)$. Therefore Example 2.1 will deterministically evaluate to 3, regardless of the order in which the two `put` operations occurred.

On the other hand, if *lv*'s lattice is that shown in Figure 2.2(b), in which the least upper bound of any two distinct positive integers is $\top$, then Example 2.1 will deterministically raise an exception, indicating that conflicting writes to *lv* have occurred. This exception is analogous to the "multiple `put`" error raised upon multiple writes to an IVar. Unlike with a traditional IVar, though, multiple writes of the *same* value (say, `put` *lv* 3 and `put` *lv* 3) will *not* raise an exception, because the least upper bound of any positive integer and itself is that integer—corresponding to the fact that multiple writes of the same value do not allow any nondeterminism to be observed.

2.2.1. Threshold reads. However, merely ensuring that writes to an LVar are monotonically increasing is not enough to ensure that programs behave deterministically. Consider again the lattice of Figure 2.2(a) for *lv*, but suppose we change Example 2.1 to allow the `get` operation to be interleaved with the two `puts`:

(Example 2.2)
$$\begin{aligned} \texttt{let par } \_ &= \texttt{put } lv\ 3 \\ \_ &= \texttt{put } lv\ 2 \\ x &= \texttt{get } lv \\ \texttt{in } x & \end{aligned}$$

Since the two `puts` and the `get` can be scheduled in any order, Example 2.2 is nondeterministic: x might be either 2 or 3, depending on the order in which the LVar effects occur. Therefore, to maintain determinism, LVars put an extra restriction on the `get` operation. Rather than allowing `get` to observe the exact value of the LVar, it can only observe that the LVar has reached one of a specified set of *lower bound* states. This set of lower bounds, which we provide as an extra argument to `get`, is called a *threshold set* because the values in it form a "threshold" that the state of the LVar must cross before the call to `get` is allowed to unblock and return. When the threshold has been reached, `get` unblocks and returns *not* the exact value of the LVar, but instead, the (unique) element of the threshold set that has been reached or surpassed.

We can make Example 2.2 behave deterministically by passing a threshold set argument to `get`:

(Example 2.3)
$$\begin{aligned} \texttt{let par } \_ &= \texttt{put } lv\ 3 \\ \_ &= \texttt{put } lv\ 2 \\ x &= \texttt{get } lv\ \{3\} \\ \texttt{in } x & \end{aligned}$$

For instance, suppose we choose the singleton set $\{3\}$ as the threshold set. Since *lv*'s value can only increase with time, we know that once it is at least 3, it will remain at or above 3 forever; therefore

the program will deterministically evaluate to 3. Had we chosen $\{2\}$ as the threshold set, the program would deterministically evaluate to 2; had we chosen $\{4\}$, it would deterministically block forever.

As long as we only access LVars with `put` and (thresholded) `get`, we can arbitrarily share them between threads without introducing nondeterminism. That is, the `put` and `get` operations in a given program can happen in any order, without changing the value to which the program evaluates.

2.2.2. Incompatibility of threshold sets. While the LVar interface just described is deterministic, it is only useful for synchronization, not for communicating data: we must specify in advance the single answer we expect to be returned from the call to `get`. In general, though, threshold sets do not have to be singleton sets. For example, consider an LVar *lv* whose states form a lattice of *pairs* of natural-number-valued IVars; that is, *lv* is a pair (m, n), where m and n both start as $\bot$ and may each be updated once with a non-$\bot$ value, which must be some natural number. This lattice is shown in Figure 2.2(c).

We can then define `getFst` and `getSnd` operations for reading from the first and second entries of *lv*:

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

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

This allows us to write programs like the following:

(Example 2.4)

$$\begin{aligned} \texttt{let par}\ \_ &= \texttt{put}\ \textit{lv}\ (\bot, 4) \\ \_ &= \texttt{put}\ \textit{lv}\ (3, \bot) \\ x &= \texttt{getSnd}\ \textit{lv} \\ \texttt{in}\ x & \end{aligned}$$

In the call `getSnd` *lv*, the threshold set is $\{(\bot, 0), (\bot, 1), \dots\}$, an infinite set. There is no risk of nondeterminism because the elements of the threshold set are *pairwise incompatible* with respect to *lv*'s lattice: informally, since the second entry of *lv* can only be written once, no more than one state

from the set $\{(\bot, 0), (\bot, 1), \dots\}$ can ever be reached. (I formalize this incompatibility requirement in Section 2.3.3.)

In the case of Example 2.4, `getSnd` *lv* may unblock and return $(\bot, 4)$ any time after the second entry of *lv* has been written, regardless of whether the first entry has been written yet. It is therefore possible to use LVars to safely read parts of an incomplete data structure—say, an object that is in the process of being initialized by a constructor.

2.2.3. The model versus reality. The use of explicit threshold sets in the LVars model should be understood as a mathematical modeling technique, *not* an implementation approach or practical API. The LVish library (which I will discuss in Chapter 4) provides an unsafe `getLV` operation to the authors of LVar data structure libraries, who can then make operations like `getFst` and `getSnd` available as a safe interface for application writers, implicitly baking in the particular threshold sets that make sense for a given data structure without ever explicitly constructing them.

To put it another way, operations on a data structure exposed as an LVar must have the *semantic effect* of a least upper bound for writes or a threshold for reads, but none of this need be visible to clients (or even written explicitly in the code). Any data structure API that provides such a semantics is guaranteed to provide deterministic concurrent communication.

2.3. Lattices, stores, and determinism

As a minimal substrate for LVars, I 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 I am interested in modeling with λ_{LVar} are those with explicit effectful operations on shared data structures, in which parallel 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.[5] 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 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, so long as 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, I 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.

2.3.1. Lattices. The definition of λ_{LVar} is parameterized by D: to write concrete λ_{LVar} programs, we must specify the set of LVar states that we are 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), written $\sqcup$. 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.

We can specify all these components as a 4-tuple $(D, \sqsubseteq, \bot, \top)$ where D is a set, $\sqsubseteq$ is a partial order on the elements of D, $\bot$ is the least element of D according to $\sqsubseteq$, and $\top$ is the greatest. However, I use D as a shorthand for the entire 4-tuple $(D, \sqsubseteq, \bot, \top)$ when its meaning is clear from the context.

[5] IVars are so named because they are a special case of *I-structures* [3]—namely, those with only one cell.

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 as union; for pointer-based data structures like tries, it could allow for unification of partially-initialized structures.

LK: Since I've already talked abut this stuff in Section 2.2, I'm kind of worried that having it here again belabors the point, although the treatment here is more formal.

The simplest example of a useful lattice is one that represents the states that a single-assignment variable (that is, an IVar) can take on. A natural-number-valued IVar, for instance, has a state space corresponding to the lattice in Figure 2.2(b), that is,

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

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 2.2(a), on the other hand, the $\top$ state is unreachable, because the least upper bound of any two writes is just the maximum of the two. 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.

2.3.2. 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* that maps from a location l, drawn from a countable set *Loc*, to a

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.[6]

Definition 2.1. A *store* is either a finite partial mapping $S : \mathit{Loc} \xrightarrow{\text{fin}} (D - \{\top\})$, or the distinguished element $\top_S$.

I use the notation $S[l \mapsto d]$ to denote extending S with a binding from l to d. If $l \in \mathit{dom}(S)$, then $S[l \mapsto d]$ denotes an update to the existing binding for l, rather than an extension. Another way to denote a store is 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 2.2 (store ordering). A store S is *less than or equal to* a store S' (written $S \sqsubseteq_S S'$) iff:

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

Definition 2.3 (least upper bound of stores). The *least upper bound* 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 $l \in \mathit{dom}(S_1) \cap \mathit{dom}(S_2)$ such that $S_1(l) \sqcup S_2(l) = \top$.
- Otherwise, $S_1 \sqcup_S S_2$ is the store S such that:
  - $\mathit{dom}(S) = \mathit{dom}(S_1) \cup \mathit{dom}(S_2)$, and
  - For all $l \in \mathit{dom}(S)$:

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

[6] In practice, different LVars in a program might have different state spaces (and, in the LVish Haskell library that I will present in Chapter 4, they do). Multiple lattices can in principle be encoded using a sum construction, so this modeling choice is just to keep the presentation simple.

Equivalence of stores is also straightforward. Two stores are equal if they are both $\top_S$ or if they both have the same set of bindings:

Definition 2.4 (equality of stores). Two stores S and S' are *equal* iff:

(1) $S = \top_S$ and $S' = \top_S$, or

(2) $\mathit{dom}(S) = \mathit{dom}(S')$ and for all $l \in \mathit{dom}(S)$, $S(l) = S'(l)$.

By Definition 2.3, 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 I will show in Section 2.4) an attempted write that would take the state of l to $\top$ would raise an error before the write can occur.

2.3.3. 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 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* T, which is a non-empty subset of D that is *pairwise incompatible*, meaning that the lub of any two distinct elements in T is $\top$. If the LVar's state d_1 in the lattice is *at or above* some $d_2 \in T$, the `get` operation unblocks and returns d_2. Note that d_2 is a unique element of T, for if there is another $d_2' \neq d_2$ in the threshold set such that $d_2' \sqsubseteq d_1$, it would follow that $d_2 \sqcup d_2' \sqsubseteq d_1$, and so $d_2 \sqcup d_2'$ cannot be $\top$, which contradicts the requirement that T 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". Figure 2.2(b) shows what the "tripwire" looks like for an example `get` operation. 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.

The requirement that the elements of a threshold set be pairwise incompatible limits the expressivity of threshold sets. In fact, it is a stronger requirement than we need to ensure determinism. Later on, in Section 2.6, I will explain how to generalize the definition of threshold sets to make more programs expressible. For now, I will proceed with the simpler definition above.

2.3.4. Monotonic store growth and determinism. In IVar-based languages, a store can only change in one of two ways: a new, empty location (pointing to $\bot$) is created, 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* [**11**], a single-assignment imperative calculus that models the Intel Concurrent Collections (CnC) system, defines ordering on stores as follows:[7]

Definition 2.5 (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 2.2 is reminiscent of Definition 2.5, but Definition 2.5 requires that $S(l)$ and $S'(l)$ be *equal*, instead of our weaker requirement that $S(l)$ be *less than or equal to* $S'(l)$ according to the application-specific lattice $\sqsubseteq$. In λ_{LVar}, stores may grow by updating existing bindings via repeated

[7] A minor difference between λ_{LVar} and Featherweight CnC is that, in Featherweight CnC, no store location is explicitly bound to $\bot$. Instead, if $l \notin dom(S)$, then l is defined to point to $\bot$.

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

$$
\begin{array}{rrcl}
\text{configurations} & \sigma & ::= & \langle S;\, e\rangle \mid \textbf{error} \\
\text{expressions} & e & ::= & x \mid v \mid e\ e \mid \texttt{get}\ e\ e \mid \texttt{put}\ e\ e \mid \texttt{new} \\
\text{values} & v & ::= & \texttt{()} \mid d \mid l \mid T \mid \lambda x.\, e \\
\text{threshold sets} & T & ::= & \{d_1,\, d_2,\, \ldots\} \\
\text{stores} & S & ::= & [l_1 \mapsto d_1,\, \ldots,\, l_n \mapsto d_n] \mid \top_S \\
\text{evaluation contexts} & E & ::= & [\,] \mid E\ e \mid e\ E \mid \texttt{get}\ E\ e \mid \texttt{get}\ e\ E \mid \texttt{put}\ E\ e \mid \texttt{put}\ e\ E
\end{array}
$$

Figure 2.3. Syntax for λ_{LVar}.

puts, so Definition 2.5 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 2.2, but would not hold under Definition 2.5. 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.

I establish in Section 2.5 that λ_{LVar} remains deterministic despite the relatively weak $\sqsubseteq_S$ relation given in Definition 2.2. 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.

2.4. λ_{LVar}: syntax and semantics

The syntax of λ_{LVar} appears in Figure 2.3, and Figures 2.4 and 2.5 together give the operational semantics. Both the syntax and semantics are parameterized by the lattice $(D, \sqsubseteq, \bot, \top)$.

A *configuration* $\langle S;\, e\rangle$ comprises a store and an expression. The *error configuration*, written **error**, is a unique element added to the set of configurations, but $\langle \top_S;\, e\rangle$ is equal to **error** for all expressions e. The metavariable σ ranges over configurations.

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

$\mathit{incomp}(T) \triangleq \forall\, d_1, d_2 \in T.\ (d_1 \neq d_2 \implies d_1 \sqcup d_2 = \top)$ $\boxed{\sigma \hookrightarrow \sigma'}$

E-Beta

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

E-New

$$\frac{}{\langle S;\ \texttt{new}\rangle \longrightarrow \langle S[l \mapsto \bot];\ l\rangle}\ (l \notin \mathit{dom}(S))$$

E-Put

$$\frac{S(l) = d_1 \qquad d_1 \sqcup d_2 \neq \top}{\langle S;\ \texttt{put}\ l\ d_2\rangle \longrightarrow \langle S[l \mapsto d_1 \sqcup d_2];\ ()\rangle}$$

E-Put-Err

$$\frac{S(l) = d_1 \qquad d_1 \sqcup d_2 = \top}{\langle S;\ \texttt{put}\ l\ d_2\rangle \longrightarrow \textbf{error}}$$

E-Get

$$\frac{S(l) = d_1 \qquad \mathit{incomp}(T) \qquad d_2 \in T \qquad d_2 \sqsubseteq d_1}{\langle S;\ \texttt{get}\ l\ T\rangle \longrightarrow \langle S;\ d_2\rangle}$$

Figure 2.4. Reduction semantics for λ_{LVar}.

$\boxed{\sigma \longmapsto \sigma'}$

E-Eval-Ctxt

$$\frac{\langle S;\ e\rangle \longrightarrow \langle S';\ e'\rangle}{\langle S;\ E[e]\rangle \longmapsto \langle S';\ E[e']\rangle}$$

Figure 2.5. Context semantics for λ_{LVar}.

Stores are as described in Section 2.3.2, and expressions may be variables, values, function applications, `get` expressions, `put` expressions, or `new`. The value forms include the unit value `()`, elements d of the specified lattice, store locations l, threshold sets T, or λ-expressions $\lambda x.\, e$. A threshold set is a set $\{d_1, d_2, \dots\}$ of one or more elements of the specified lattice.

The operational semantics is split into two parts, a *reduction semantics*, shown in Figure 2.4, and a *context semantics*, shown in Figure 2.5.

The reduction relation $\longrightarrow$ is defined on configurations. There are five rules in the reduction semantics: the E-Beta rule is standard β-reduction, and the rules E-New, E-Put/E-Put-Err, and E-Get respectively express the semantics of the `new`, `put`, and `get` operations described in Section 2.3.3. The E-New rule creates a new binding in the store and returns a pointer to it; the side condition $l \notin \mathit{dom}(S)$ ensures that l is a fresh location. The E-Put rule updates the store and returns `()`, the unit value. The E-Put-Err rule applies when a `put` to a location would take its state to $\top$; in that case, the semantics steps to **error**. The incompatibility of the threshold set argument to `get` is enforced in the E-Get rule by the $\mathit{incomp}(T)$ premise, which requires that the least upper bound of any two distinct elements in T must be $\top$.[8]

The context relation $\longmapsto$ is also defined on configurations. It has only one rule, E-Eval-Ctxt, which is a standard context rule, allowing reductions to apply within a context. The choice of context determines where evaluation can occur; in λ_{LVar}, the order of evaluation is nondeterministic (that is, a given expression can generally reduce in more than one way), and so it is generally *not* the case that an expression has a unique decomposition into redex and context.[9] For example, in an application $e_1\ e_2$, either e_1 or e_2 might reduce first. The nondeterminism in choice of evaluation context reflects the nondeterminism of scheduling between concurrent threads, and in λ_{LVar}, the arguments to `get`, `put`, and application expressions are *implicitly* evaluated concurrently.

2.4.1. Errors and observable determinism. Is the `get` operation deterministic? Consider two lattice elements d_1 and d_2 that have no ordering and have $\top$ as their lub, and suppose that `puts` of d_1 and d_2 and a `get` with $\{d_1, d_2\}$ as its threshold set all race for access to an LVar lv. Eventually, the program

[8] Although $\mathit{incomp}(T)$ is given as a premise of the E-Get reduction rule (suggesting that it is checked at runtime), as I noted earlier in Section 2.2.3, in a real implementation of LVars threshold sets need not have any runtime representation, nor do they need to be written explicitly in the code. Rather, it is the data structure author's responsibility to ensure that any operations provided for reading from LVars have threshold semantics.

[9] In fact, my motivation for splitting the operational semantics into a reduction semantics and a context semantics is to isolate the nondeterminism of the context semantics, which simplifies the determinism proof of Section 2.5.

is guaranteed to raise **error** by way of the E-Put-Err rule, because $d_1 \sqcup d_2 = \top$. Before that happens, though, `get` lv $\{d_1, d_2\}$ could return either d_1 or d_2. Therefore, `get` *can* behave nondeterministically—but this behavior is not observable in the final outcome of the program, and under our definition of observable determinism, only the final outcome of a program counts as what is observable.

2.4.2. Fork-join parallelism. λ_{LVar} has a call-by-value semantics: arguments must be fully evaluated before function application (β-reduction, via the E-Beta rule) can occur. I 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:

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

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 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 [22].

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 2.6(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 2.6(b). This adds the ability to construct arbitrary non-series-parallel dependency graphs, just as with *first-class futures* [46].

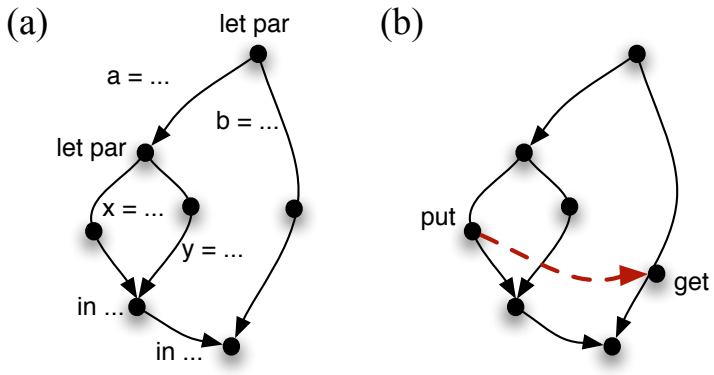

Figure 2.6. A series-parallel graph induced by parallel λ-calculus evaluation (a); a non-series-parallel graph induced by `put`/`get` operations (b).

Because we do not reduce under λ-terms, we can sequentially compose e_1 before e_2 by writing $\texttt{let \_} = e_1 \texttt{ in } e_2$, which desugars to $(\lambda\_.\, e_2)\, e_1$. Sequential composition is useful for situations when expressions must run in a particular order, *e.g.*, if we want to first allocate a new LVar with `new` and then call `put` on it.

2.5. Proof of determinism for λ_{LVar}

The main technical result of this chapter is a proof of determinism for the λ_{LVar} language. The determinism theorem says that if two executions starting from a given configuration σ terminate in configurations σ' and σ'', then σ' and σ'' are the same configuration, up to a permutation on locations. (I discuss permutations in more detail below, in Section 2.5.1.)

In order to prove determinism for λ_{LVar}, I first prove several supporting lemmas. Lemma 2.1 (Permutability) deals with location names, and Lemma 2.3 (Locality) establishes a useful property for dealing with expressions that decompose into redex and context in multiple ways. After that point, the structure of the proof is similar to that of the proof of determinism for Featherweight CnC given by Budimlić *et al.* [11]. I reuse the naming conventions of Budimlić *et al.* for Lemmas 2.4 (Monotonicity), 2.5 (Independence), 2.6 (Clash), 2.7 (Error Preservation), and 2.8 (Strong Local Confluence). However, the statements

and proofs of those properties differ considerably in the setting of λ_{LVar}, due to the generality of LVars and other differences between the λ_{LVar} language and Featherweight CnC.

On the other hand, Lemmas 2.9 (Strong One-Sided Confluence), 2.10 (Strong Confluence), and 2.11 (Confluence) are nearly identical to the corresponding lemmas in the Featherweight CnC determinism proof. This is the case because, once Lemmas 2.4 through 2.8 are established, the remainder of the determinism proof does not need to deal specifically with the semantics of LVars, lattices, or the store, and instead deals only with execution steps at a high level.

2.5.1. Permutations and permutability. The E-New rule allocates a fresh location $l \in \mathit{Loc}$ in the store, with the only requirement on l being that it is not (yet) in the domain of the store. Therefore, multiple runs of the same program may differ in what locations they allocate, and therefore the reduction semantics is nondeterministic with respect to allocation. However, this is not a kind of nondeterminism that we care about, so we work modulo an arbitrary *permutation* on locations.

Recall from Section 2.3.2 that we have a countable set of locations *Loc*. Then, a permutation is defined as follows:

Definition 2.6 (permutation). A *permutation* is a function $\pi : \mathit{Loc} \to \mathit{Loc}$ such that:

(1) it is invertible, that is, there is an inverse function $\pi^{-1} : \mathit{Loc} \to \mathit{Loc}$ with the property that $\pi(l) = l'$ iff $\pi^{-1}(l') = l$; and

(2) it is the identity on all but finitely many elements of Loc.

Condition (1) in Definition 2.6 ensures that we only consider location renamings that we can "undo", and condition (2) ensures that we only consider renamings of a finite number of locations. Equivalently, we can say that π is a bijection from *Loc* to *Loc* such that it is the identity on all but finitely many elements.

We can straightforwardly lift Definition 2.6 to expressions and stores, and hence also to configurations. To lift a permutation π to expressions, we over the syntax of terms structurally and apply π to any

locations that occur in the term. We can also lift π to evaluation contexts, structurally: $\pi([\,]) = [\,]$, $\pi(E\ e = \pi(E)\ \pi(e)$, and so on.

To lift π to stores, we apply π to all locations in the domain of the store. (We do not have to do any renaming in the codomain of the store, since locations cannot occur in elements of the application-specific lattice D.) Since π is a bijection, it follows that if some location l is not in the domain of some store S, then $\pi(l) \notin \mathit{dom}((\pi(S))$,LK: Do I need to spell out why this is true? a fact that will be useful to us shortly.

Definition 2.7 (permutation (expressions)). A *permutation* of an expression e is a function π defined as follows:

$$\begin{aligned}
\pi(x) &= x \\
\pi(\texttt{()}) &= \texttt{()} \\
\pi(d) &= d \\
\pi(l) &= \pi(l) \\
\pi(T) &= T \\
\pi(\lambda x.\, e) &= \lambda x.\, \pi(e) \\
\pi(e_1\ e_2) &= \pi(e_1)\ \pi(e_2) \\
\pi(\texttt{get}\ e_1\ e_2) &= \texttt{get}\ \pi(e_1)\ \pi(e_2) \\
\pi(\texttt{put}\ e_1\ e_2) &= \texttt{put}\ \pi(e_1)\ \pi(e_2) \\
\pi(\texttt{new}) &= \texttt{new}
\end{aligned}$$

LK: It's fine to just say that $\pi(\texttt{new})$ is new; we only care about renaming it if it has already been allocated! If it's just an unevaluated new expression, then there's nothing to do.

Definition 2.8 (permutation (stores)). A *permutation* of a store S is a function π defined as follows:

$$\begin{aligned}
\pi(\top_S) &= \top_S \\
\pi([l_1 \mapsto d_1, \ldots, l_n \mapsto d_n]) &= [\pi(l_1) \mapsto d_1, \ldots, \pi(l_n) \mapsto d_n]
\end{aligned}$$

Definition 2.9 (permutation (configurations)). A *permutation* of a configuration $\langle S;\ e\rangle$ is a function π defined as follows: if $\langle S;\ e\rangle = \textbf{error}$, then $\pi(\langle S;\ e\rangle) = \textbf{error}$; otherwise, $\pi(\langle S;\ e\rangle) = \langle \pi(S);\ \pi(e)\rangle$.

With these definitions in place, I can prove a lemma that says that the names of locations in a configuration do not affect whether or not it can take a step. Lemma 2.1 says that a configuration σ can step to σ' exactly when $\pi(\sigma)$ can step to $\pi(\sigma')$.

Lemma 2.1 (Permutability). *For any finite permutation π,*

(1) $\sigma \longhookrightarrow \sigma'$ *if and only if* $\pi(\sigma) \longhookrightarrow \pi(\sigma')$.

(2) $\sigma \longmapsto \sigma'$ *if and only if* $\pi(\sigma) \longmapsto \pi(\sigma')$.

Proof. See Section A.1. The forward direction of part 1 is by cases on the rule in the reduction semantics by which σ steps to σ'; the only interesting case is the E-New case, in which we make use of the fact that if $l \notin \mathit{dom}(S)$, then $\pi(l) \notin \mathit{dom}(\pi(S))$. The reverse direction of part 1 relies on the fact that if π is a permutation, then π^{-1} is also a permutation. Part 2 of the proof builds on part 1. □

2.5.2. Internal determinism of the reduction semantics. LK: I'm leaving this section in for the time being, but I don't actually use this fact anywhere yet! Could I have used it to simplify the proof of Lemma 2.8 or something?

My goal is to show that λ_{LVar} is deterministic according to the definition of *observable determinism* that I gave in Chapter 1—that is, that a λ_{LVar} program always evaluates to the same value. In the context of λ_{LVar}, a "program" can be understood as a configuration, and a "value" can be understood as a configuration that cannot step, either because the expression in that configuration is actually a value, or because it is a "stuck" configuration that cannot step because no rule of the operational semantics applies. In λ_{LVar}, the latter situation could occur if, for instance, a configuration contains a blocking `get` expression and there are no other expressions left to evaluate that might cause it to unblock.

This definition of observable determinism does *not* require that a configuration takes the same sequence of steps on the way to reaching its value at the end of every run, and in fact, the λ_{LVar} operational semantics does not have that property. Borrowing terminology from Blelloch *et al*. [6], I will use the term *internally deterministic* to describe a program that does, in fact, take the same sequence of steps on every run.[10] Although λ_{LVar} is not internally deterministic, all of its internal nondeterminism is due to the E-Eval-Ctxt rule! This is the case because the E-Eval-Ctxt rule is the only rule in the operational semantics by which a particular configuration can step in multiple ways. The multiple ways in which a configuration can step via E-Eval-Ctxt correspond to the ways in which the expression in that configuration can be decomposed into a redex and an evaluation context. In fact, it is exactly this property that makes it possible for multiple subexpressions of a λ_{LVar} expression (a `let par` expression, for instance) to be evaluated in parallel.

But, if we leave aside evaluation contexts for the moment (we'll return to them in the following section) and focus on only the rules of the reduction semantics in Figure 2.4, we see that there is only one rule by which a given configuration can step, and only one configuration to which it can step. The exception is the E-New rule, which nondeterministically allocates locations and returns pointers to them—but we can account for this by saying that the reduction semantics is internally deterministic up to a permutation on locations. Lemma 2.2 formalizes this claim.

Lemma 2.2 (Internal Determinism of the Reduction Semantics). *If* $\sigma \longhookrightarrow \sigma'$ *and* $\sigma \longhookrightarrow \sigma''$*, then there is a permutation* π *such that* $\sigma' = \pi(\sigma'')$.

Proof. Straightforward by cases on the rule of the reduction semantics by which σ steps to σ'; the only interesting case is for the E-New rule. See Section A.2. □

[10] I am using "internally deterministic" in a more specific way than Blelloch *et al*.: they define an internally deterministic program to be one for which the *trace* of the program is the same on every run, where a trace is a directed acyclic graph of the operations executed by the program and the control dependencies among them. This definition, in turn, depends on the definition of "operation", which might be defined in a fine-grained way or a coarse-grained, abstract way, depending on which aspects of program execution one wants the notion of internal determinism to capture. The important point is that internal determinism is a stronger property than observable determinism.

2.5.3. Locality. In order to prove determinism for λ_{LVar}, we will have to consider situations in which we have an expression that decomposes into redex and context in multiple ways. Suppose that we have an expression e such that $e = E_1[e_1] = E_2[e_2]$. The configuration $\langle S;\ e\rangle$ can then step in two different ways by the E-Eval-Ctxt rule: $\langle S;\ E_1[e_1]\rangle \longmapsto \langle S_1;\ E_1[e_1']\rangle$, and $\langle S;\ E_2[e_2]\rangle \longmapsto \langle S_2;\ E_2[e_2']\rangle$.

The key observation we can make here is that the $\longmapsto$ relation acts "locally". That is, when e_1 steps to e_1' within its context, the expression e_2 will be left alone, because it belongs to the context. Likewise, when e_2 steps to e_2' within its context, the expression e_1 will be left alone. Lemma 2.3 formalizes this claim.

Lemma 2.3 (Locality). *If* $\langle S;\ E_1[e_1]\rangle \longmapsto \langle S_1;\ E_1[e_1']\rangle$ *and* $\langle S;\ E_2[e_2]\rangle \longmapsto \langle S_2;\ E_2[e_2']\rangle$ *and* $E_1[e_1] = E_2[e_2]$*, then there exist evaluation contexts* E_1' *and* E_2' *such that:*

- $E_1'[e_1] = E_2[e_2']$*, and*
- $E_2'[e_2] = E_1[e_1']$*, and*
- $E_1'[e_1'] = E_2'[e_2']$.

Proof. TODO: Add short description here once the proof is done. See Section A.3. □

2.5.4. Monotonicity. The Monotonicity lemma says that, as evaluation proceeds according to the $\hookrightarrow$ relation, the store can only grow with respect to the $\sqsubseteq_S$ ordering.

Lemma 2.4 (Monotonicity). *If* $\langle S;\ e\rangle \hookrightarrow \langle S';\ e'\rangle$*, then* $S \sqsubseteq_S S'$.

Proof. Straightforward by cases on the rule of the reduction semantics by which $\langle S;\ e\rangle$ steps to $\langle S';\ e'\rangle$. The interesting cases are for the E-New and E-Put rules. See Section A.4. □

2.5.5. Independence. The Independence lemma establishes a "frame property" for λ_{LVar} that captures the idea that independent effects commute with each other. Consider an expression e that runs starting in store S and steps to e', updating the store to S'. The Independence lemma provides a double-edged

guarantee about what will happen if we evaluate e starting from a larger store $S \sqcup_S S''$: first, e will update the store to $S' \sqcup_S S''$; second, e will step to e' as it did before. Here $S \sqcup_S S''$ is the least upper bound of the original S and some other store S'' that is "framed on" to S; intuitively, S'' is the store resulting from some other independently-running computation.

Lemma 2.5 requires as a precondition that the store S'' must be *non-conflicting* with the original transition from $\langle S;\ e\rangle$ to $\langle S';\ e'\rangle$, meaning that locations in S'' cannot share names with locations newly allocated during the transition; this rules out location name conflicts caused by allocation.

Definition 2.10 (non-conflicting store). A store S'' is *non-conflicting* with the transition $\langle S;\ e\rangle \longrightarrow \langle S';\ e'\rangle$ iff $(\mathit{dom}(S') - \mathit{dom}(S)) \cap \mathit{dom}(S'') = \emptyset$.

Lemma 2.5 (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. By cases on the rule of the reduction semantics by which $\langle S;\ e\rangle$ steps to $\langle S';\ e'\rangle$. The interesting cases are for the E-New and E-Put rules. Since $\langle S';\ e'\rangle \neq$ **error**, we do not need to consider the E-Put-Err rule. See Section A.5. □

2.5.6. Clash. The Clash lemma, Lemma 2.6, is similar to the Independence lemma, but handles the case where $S' \sqcup_S S'' = \top_S$. It establishes that, in that case, $\langle S \sqcup_S S'';\ e\rangle$ steps to **error**.

Lemma 2.6 (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 $S' \sqcup_S S'' = \top_S$:*

$\langle S \sqcup_S S'';\ e\rangle \longrightarrow^i$ **error**, *where $i \leq 1$.*

Proof. By cases on the rule of the reduction semantics by which $\langle S;\ e\rangle$ steps to $\langle S';\ e'\rangle$. As with Lemma 2.5, the interesting cases are for the E-New and E-Put rules, and since $\langle S';\ e'\rangle \neq$ **error**, we do not need to consider the E-Put-Err rule. See Section A.6. □

2.5.7. Error Preservation. Lemma 2.7, Error Preservation, says that if a configuration $\langle S;\ e\rangle$ steps to **error**, then evaluating e in the context of some larger store will also result in **error**.

Lemma 2.7 (Error Preservation). *If* $\langle S;\ e\rangle \longrightarrow$ **error** *and* $S \sqsubseteq_S S'$*, then* $\langle S';\ e\rangle \longrightarrow$ **error**.

Proof. Suppose $\langle S;\ e\rangle \longrightarrow$ **error** and $S \sqsubseteq_S S'$. We are required to show that $\langle S';\ e\rangle \longrightarrow$ **error**.

By inspection of the operational semantics, the only rule by which $\langle S;\ e\rangle$ can step to **error** is E-Put-Err. Hence $e = \texttt{put}\ l\ d_2$. From the premises of E-Put-Err, we have that $S(l) = d_1$. Since $S \sqsubseteq_S S'$, it must be the case that $S'(l) = d_1'$, where $d_1 \sqsubseteq d_1'$. Since $d_1 \sqcup d_2 = \top$, we have that $d_1' \sqcup d_2 = \top$. Hence, by E-Put-Err, $\langle S';\ \texttt{put}\ l\ d_2\rangle \longrightarrow$ **error**, as we were required to show. □

2.5.8. Confluence. Lemma 2.8, the Strong Local Confluence lemma, says that if a configuration σ can step to configurations σ_a and σ_b, then there exists an configuration σ_c that σ_a and σ_b can each reach in at most one step, modulo a permutation on the locations in σ_b. Lemmas 2.9 and 2.10 then generalize that result to arbitrary numbers of steps.

The structure of this part of the proof differs slightly from the Budimlić *et al*. determinism proof for Featherweight CnC. Budimlić *et al*. prove a *diamond* property, in which σ_a and σ_b each step to σ_c in *exactly* one step. They then get a property like Lemma 2.8 as an immediate consequence of the diamond property, by choosing $i = j = 1$. But a true diamond property with exactly one step "on each side of the diamond" is stronger than we need here, and, in fact, does not hold for λ_{LVar}; so, instead, I prove the weaker "at most one step" property directly.

Lemma 2.8 (Strong Local Confluence). *If* $\sigma \longmapsto \sigma_a$ *and* $\sigma \longmapsto \sigma_b$*, then there exist* σ_c, i, j, π *such that* $\sigma_a \longmapsto^i \sigma_c$ *and* $\pi(\sigma_b) \longmapsto^j \sigma_c$ *and* $i \leq 1$ *and* $j \leq 1$.

Proof. Since the original configuration σ can step in two different ways, its expression decomposes into redex and context in two different ways: $\sigma = \langle S;\ E_a[e_{a_1}]\rangle = \langle S;\ E_b[e_{b_1}]\rangle$, where $E_a[e_{a_1}] = E_b[e_{b_1}]$,

but E_a and E_b may differ and e_{a_1} and e_{b_1} may differ. We can then apply the Lemma 2.3 (Locality) lemma; at a high level, it shows that e_{a_1} and e_{b_1} can step independently within their contexts.

The proof is then by a double case analysis on the rule of the reduction semantics by which e_{a_1} steps and by which e_{b_1} steps. In order to combine the results of the two steps, the proof makes use of Lemma 2.5 (Independence). The most interesting case is that in which both expressions step by the E-New rule and they allocate locations with the same name. In that case, we can use Lemma 2.1 (Permutability) to rename locations so as not to conflict. See Section A.7. □

Lemma 2.9 (Strong One-Sided Confluence). *If $\sigma \longmapsto \sigma'$ and $\sigma \longmapsto^m \sigma''$, where $1 \leq m$, then there exist σ_c, i, j, π such that $\sigma' \longmapsto^i \sigma_c$ and $\pi(\sigma'') \longmapsto^j \sigma_c$ and $i \leq m$ and $j \leq 1$.*

Proof. By induction on m; see Section A.8. □

Lemma 2.10 (Strong Confluence). *If $\sigma \longmapsto^n \sigma'$ and $\sigma \longmapsto^m \sigma''$, where $1 \leq n$ and $1 \leq m$, then there exist σ_c, i, j, π such that $\sigma' \longmapsto^i \sigma_c$ and $\pi(\sigma'') \longmapsto^j \sigma_c$ and $i \leq m$ and $j \leq n$.*

Proof. By induction on n; see Section A.9. □

Lemma 2.11 (Confluence). *If $\sigma \longmapsto^* \sigma'$ and $\sigma \longmapsto^* \sigma''$, then there exist σ_c and π such that $\sigma' \longmapsto^* \sigma_c$ and $\pi(\sigma'') \longmapsto^* \sigma_c$.*

Proof. Strong Confluence (Lemma 2.10) implies Confluence. □

2.5.9. Determinism. Finally, the determinism theorem, Theorem 2.1, is a direct result of Lemma 2.11:

Theorem 2.1 (Determinism). *If $\sigma \longmapsto^* \sigma'$ and $\sigma \longmapsto^* \sigma''$, and neither σ' nor σ'' can take a step, then there exists π such that $\sigma' = \pi(\sigma'')$.*

Proof. We have from Lemma 2.11 that there exists σ_c and π such that $\sigma' \longmapsto^* \sigma_c$ and $\pi(\sigma'') \longmapsto^* \sigma_c$. Since σ' cannot step, we must have $\sigma' = \sigma_c$.

By Lemma 2.1 (Permutability), σ'' can step iff $\pi(\sigma'')$ can step, so since σ'' cannot step, $\pi(\sigma'')$ cannot step either.

Hence we must have $\pi(\sigma'') = \sigma_c$. Since $\sigma' = \sigma_c$ and $\pi(\sigma'') = \sigma_c$, $\sigma' = \pi(\sigma'')$. □

2.5.10. Discussion: termination. I have followed Budimlić *et al*. [11] 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 2.1 says that if two runs of a given λ_{LVar} program reach configurations where no more reductions are possible, then they have reached the same configuration. Hence Theorem 2.1 handles the case of *deadlocks* already: a λ_{LVar} program can deadlock (*e.g.*, with a blocked `get`), but it will do so deterministically.

However, Theorem 2.1 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. I conjecture (but do not prove) that such a consistent termination property holds for λ_{LVar}. Such a property could be paired with Theorem 2.1 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, the existing determinism theorem does not have to treat **error** specially.)

2.6. Generalizing the `put` and `get` operations

The determinism result for λ_{LVar} shows that adding LVars (with their accompanying `new`/`put`/`get` operations) to an existing deterministic parallel language (the λ-calculus) preserves determinism. But it is not the case that the `put` and `get` operations are *the most general* determinism-preserving operations on LVars. In this section, I consider some alternative semantics for `put` and `get` that generalize their behavior while retaining the determinism of the model.

2.6.1. Generalizing from least-upper-bound writes to inflationary, commutative writes. In the LVars model as presented in this chapter so far, the only way for the state of an LVar to evolve over time is through a series of least-upper-bound updates resulting from `put` operations. Unfortunately, this way of updating an LVar provides no efficient way to model, for instance, an atomically incremented counter that occupies one memory location. Consider an LVar based on the lattice of Figure 2.2(c). Under the least-upper-bound semantics of `put`, if two independent writes each take the LVar's contents from, say, 1 to 2, then after both writes, its contents will be 2, because `put` takes the maximum of the previous value and the current value. Although this semantics is deterministic, it is not the desired semantics for every application. Instead, we might want each write to *increment* the contents of the LVar by one, resulting in 3.

LK: "Although it is tempting to think that LVar writes are always idempotent, there is a `Counter` example."

To support this alternative semantics in the LVars model, we generalize the model as follows. For an LVar with lattice $(D, \sqsubseteq, \bot, \top)$, we can define a set of *update* operations $u_i : D \to D$, which must meet the following two conditions:

- $\forall d, i.\ \ d \sqsubseteq u_i(d)$
- $\forall d, i, j.\ \ u_i(u_j(d)) = y_j(u_i(d))$

The first of these conditions says that each update operation is inflationary with respect to $\sqsubseteq$. The second condition says that update operations commute with each other. These two conditions correspond to the two informal criteria that we set forth for monotonic data structures at the beginning of this chapter: the requirement that updates be inflationary corresponds to the fact that monotonic data structures can only "grow", and the requirement that updates be commutative corresponds to the fact that the order of updates must not be observable.[11]

[11] Of course, commutativity of updates alone is not enough to assure that the order of updates is not observable; for that we also need threshold reads.

LK: "update" used to be called "bump", but I think "update" is a better name because I want it to be clear that `put` is a special case of it, and "bump" sounds like something that is specifically *not* idempotent, while "update" is merely something that *doesn't have to be* idempotent.

In fact, the `put` operation meets the above two conditions, and therefore can be viewed as a special case of an update operation that, in addition to being inflationary and commutative, also happens to compute a least upper bound. However, when generalizing LVars to support update operations, we must keep in mind that least-upper-bound `put` operations do not necessarily mix with arbitrary update operations on the same LVar. For example, consider a set of update operations $\{u_{(+1)}, u_{(+2)}, \dots\}$ for atomically incrementing a counter represented by a natural number LVar, with a lattice ordered by the usual $\leq$ on natural numbers. The $u_{(+1)}$ operation increments the counter by one, $u_{(+2)}$ increments it by two, and so on. It is easy to see that these operations commute. However, a `put` of 4 and a $u_{(+1)}$ do not commute: if we start with an initial state of 0 and the `put` occurs first, then the state of the LVar changes to 4 since $\max(0, 4) = 4$, and the subsequent $u_{(+1)}$ updates it to 5. But if the $u_{(+1)}$ happens first, then the final state of the LVar will be $\max(1, 4) = 4$. Furthermore, multiple distinct families of update operations only commute among themselves and cannot be combined.

In practice, the author of a particular LVar data structure must choose which update operations that data structure should provide, and it is the data structure author's responsibiity to ensure that they commute. For example, the LVish Haskell library (which I discuss in Chapter 4) provides a set data structure, `Data.LVar.Set`, that supports only `put`, whereas the counter data structure `Data.LVar.Counter` supports only increments; an attempt to call `put` on a `Counter` would be ruled out by the type system. However, *composing* LVars that support different families of update operations is fine. For example, an LVar could represent a monotonically growing collection (which supports `put`) of counter LVars, where each counter is itself monotonically increasing and supports only increment. Indeed, the PhyBin case study described in Section 4.6 uses just such a collection of counters.[12]

[12]Unlike `put`, arbitrary update operations are not idempotent, which means that in order to support arbitrary update operations we must be careful not to assume idempotency of writes in the LVish scheduler implementation. I discuss this consideration in more detail in Section 4.4.

The determinism of λ_{LVar} relies on the fact that the states of all LVars evolve monotonically with respect to their lattices, and that the least upper bound operation is commutative and therefore the order in which `puts` occur does not matter. Together, these properties suffice to ensure that the threshold reads made by `get` operations are deterministic. The rest of a λ_{LVar} program is purely functional, and its behavior is, in fact, a pure function of these `get` observations. Since a given family of update operations is also commutative and inflationary with respect to the lattice of the LVar they operate on, they preserve determinism.

In Chapter 3, in addition to extending λ_{LVar} to support the new features of *freezing* and *event handlers*, I generalize the `put` operation to allow arbitrary update operations. The resulting generalized language definition is therefore parameterized not only by an application-specific lattice, but also by an application-specific set of commutative and inflationary update operations.

LK: The Chapter 3 λ_{LVish} result is quasi-determinism, not determinism. So, if I really wanted to be serious about showing that update is deterministic and not merely quasi-deterministic, I would do this more modularly: I would actually present another version of the λ_{LVar} language here, with `put` replaced with update, and then re-do the determinism proof for it, instead of just lumping it into the quasi-determinism result in the next chapter. If I did that, though, then this section would be big enough to justify being its own chapter. I'm going to just appeal to people's common sense that the quasi-determinism in λ_{LVish} comes from freezing, not from update. I hope that this is OK!

2.6.2. A more general formulation of threshold sets. Certain deterministic computations are difficult to express using the definition of threshold sets presented in Section 2.3.3. For instance, consider an LVar that stores the result of a parallel logical "and" operation on two Boolean inputs. I'll call this data structure an AndLV, and its two inputs the *left* and *right* inputs, respectively.

We can represent the states an AndLV can take on as pairs (x, y), where each of x and y are `T`, `F`, or `Bot`. The $(\texttt{Bot}, \texttt{Bot})$ state is the state in which no input has yet been received, and so it is the least element in the lattice of states that our AndLV can take on, shown in Figure 2.7. An additional state,

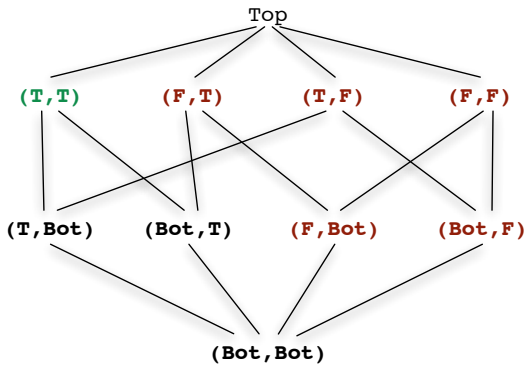

Figure 2.7. Lattice of states that an AndLV can take on. The five red states in the lattice correspond to a false result, and the one green state corresponds to a true one.

Top, is the greatest element of the lattice; it represents the situation in which an error has occurred—if, for instance, one of the inputs writes T and then later changes its mind to F.

The lattice induces a lub operation on pairs of states; for instance, the lub of $(\texttt{T}, \texttt{Bot})$ and $(\texttt{Bot}, \texttt{F})$ is $(\texttt{T}, \texttt{F})$, and the lub of $(\texttt{T}, \texttt{Bot})$ and $(\texttt{F}, \texttt{Bot})$ is Top since the overlapping T and F values conflict. As usual for LVars, the put operation updates the AndLV's state to the lub of the incoming state and the current state.

We are interested in learning whether the result of our parallel "and" computation is "true" or "false". Let's consider what observations it is possible to make of an AndLV under our existing definition of threshold reads. The states $(\texttt{T}, \texttt{T})$, $(\texttt{T}, \texttt{F})$, $(\texttt{F}, \texttt{T})$, and $(\texttt{F}, \texttt{F})$ are all pairwise incompatible with one another, and so $\{(\texttt{T}, \texttt{T}), (\texttt{T}, \texttt{F}), (\texttt{F}, \texttt{T}), (\texttt{F}, \texttt{F})\}$—that is, the set of states in which both the left and right inputs have arrived—is a legal threshold set argument to get. The trouble with this threshold read is that it does not allow us to get *early answers* from the computation. It would be preferable to have a get operation that would "short circuit" and unblock immediately if a single input of, say, $(\texttt{F}, \texttt{Bot})$ or $(\texttt{Bot}, \texttt{F})$ was written, since no later write could change the fact that the result of the whole computation would be "false".[13] Unfortunately, we cannot include $(\texttt{F}, \texttt{Bot})$ or $(\texttt{Bot}, \texttt{F})$ in our threshold set, because

[13] Actually, this is not quite true: a write of $(\texttt{F}, \texttt{Bot})$ followed by a write of $(\texttt{T}, \texttt{Bot})$ would lead to a result of Top, and to the program stepping to the **error** state, which is certainly different from a result of "false". But, even if a write of $(\texttt{T}, \texttt{Bot})$

the resulting threshold set would no longer be pairwise incompatible, and therefore would compromise determinism.

In order to get short-circuiting behavior from an AndLV without compromising determinism, we need to make a slight generalization to how threshold sets and threshold reads work. In the new formulation, we divide up threshold sets into subsets that we call *activation sets*, each consisting of *activation states*. In the case of the observation we want to make of our AndLV, one of those activation sets is the set of states that the data structure might be in when a state containing at least one F value has been written—that is, the set $\{(\texttt{F}, \texttt{Bot}), (\texttt{Bot}, \texttt{F}), (\texttt{F}, \texttt{T}), (\texttt{T}, \texttt{F}), (\texttt{F}, \texttt{F})\}$. When we reach a point in the lattice that is at or above any of those states, we know that the result will be "false". The other activation set is the singleton set $\{(\texttt{T}, \texttt{T})\}$, since we have to wait until we reach the state $(\texttt{T}, \texttt{T})$ to know that the result is "true"; a state like $(\texttt{T}, \texttt{Bot})$ does not appear in any of our activation sets.

We can now redefine "threshold set" to mean *a set of activation sets*. Under this definition, the entire threshold set that we would use to observe the contents of our AndLV is:

$$\{\{(\texttt{F}, \texttt{Bot}), (\texttt{Bot}, \texttt{F}), (\texttt{F}, \texttt{T}), (\texttt{T}, \texttt{F}), (\texttt{F}, \texttt{F})\}, \{(\texttt{T}, \texttt{T})\}\}$$

We redefine the semantics of `get` as follows: if an LVar's state reaches (or surpasses) any state or states in a particular activation set in the threshold set, `get` returns *that entire activation set*, regardless of which of its activation states was reached. If no state in any activation set in the threshold set has yet been reached, the `get` operation will block. In the case of our AndLV, as soon as either input writes a state containing an F, our `get` will unblock and return the first activation set, that is, $\{(\texttt{F}, \texttt{Bot}), (\texttt{Bot}, \texttt{F}), (\texttt{F}, \texttt{T}), (\texttt{T}, \texttt{F}), (\texttt{F}, \texttt{F})\}$. Hence AndLV has the expected "short-circuit" behavior and does not have to wait for a second input if the first input contains an F. If, on the other hand, the inputs are $(\texttt{T}, \texttt{Bot})$ and $(\texttt{Bot}, \texttt{T})$, the `get` will unblock and return $\{(\texttt{T}, \texttt{T})\}$.

---

is due to come along sooner or later to take the state of the AndLV to Top and thus raise **error**, it should still be fine for the `get` operation to allow "short-circuit" unblocking, because the result of the `get` operation does not count as observable under our definition of observable determinism (as discussed in Section 2.4.1).

In a real implementation, of course, the value returned from the `get` could be more meaningful to the client—for instance, a Haskell implementation could return `False` instead of returning the actual activation set that corresponds to "false". However, the translation from $\{(\mathtt{F}, \mathtt{Bot}), (\mathtt{Bot}, \mathtt{F}), (\mathtt{F}, \mathtt{T}), (\mathtt{T}, \mathtt{F}), (\mathtt{F}, \mathtt{F})\}$ to `False` could just as easily take place on the client side. In either case, the activation set returned from the threshold read is the same regardless of *which* of its activation states caused the read to unblock, and it is impossible for the client to tell whether the actual state of the lattice is, say, $(\mathtt{T}, \mathtt{F})$, $(\mathtt{F}, \mathtt{F})$, or some other state containing F.

As part of this activation-set-based formulation of threshold sets, we need to adjust our criterion for pairwise incompatibility of threshold sets. Recall that the purpose of the pairwise incompatibility requirement (see Section 2.3.3) was to ensure that a threshold read would return a unique result. We need to generalize this requirement, since although more than one element *in the same activation set* might be reached or surpassed by a given write to an LVar, it is still the case that writes should only unblock a *unique* activation set in the threshold set. The pairwise incompatibility requirement then becomes that elements in an activation set must be *pairwise incompatible* with elements in every other activation set. That is, for all distinct activation sets Q and R in a given threshold set:

$$\forall q \in Q. \, \forall r \in R. \, q \sqcup r = \top$$

In our AndLV example, there are two distinct activation sets, so if we let $Q = \{(\mathtt{T}, \mathtt{T})\}$ and $R = \{(\mathtt{F}, \mathtt{Bot}), (\mathtt{Bot}, \mathtt{F}), (\mathtt{F}, \mathtt{T}), (\mathtt{T}, \mathtt{F}), (\mathtt{F}, \mathtt{F})\}$, the least upper bound of $(\mathtt{T}, \mathtt{T})$ and r must be `Top`, where r is any element of R. We can easily verify that this is the case. Furthermore, since the lattice of states that an AndLV can take on is finite, the join function can be verified to compute a least upper bound.

To illustrate why we need pairwise incompatibility to be defined this way, consider the following (illegal) "threshold set" that does not meet the pairwise incompatibility criterion:

$$\{\{(\mathtt{F}, \mathtt{Bot}), (\mathtt{Bot}, \mathtt{F})\}, \{(\mathtt{T}, \mathtt{Bot}), (\mathtt{Bot}, \mathtt{T})\}\}$$

A threshold read corresponding to this so-called threshold set will unblock and return $\{(\texttt{F}, \texttt{Bot}), (\texttt{Bot}, \texttt{F})\}$ as soon as a state containing an F is reached, and $\{(\texttt{T}, \texttt{Bot}), (\texttt{Bot}, \texttt{T})\}$ as soon as a state containing a T is reached. If, for instance, the left input writes $(\texttt{F}, \texttt{Bot})$ and the right input writes $(\texttt{Bot}, \texttt{T})$, and these writes occur in arbitrary order, the threshold read will return a nondeterministic result, depending on the order of the two writes. But, with the properly pairwise-incompatible threshold set above, the threshold read will block until the write of $(\texttt{F}, \texttt{Bot})$ arrives, and then will deterministically return the "false" activation set, regardless of whether the write of $(\texttt{Bot}, \texttt{T})$ has arrived yet. Hence "short-circuit" evaluation is possible.

Finally, we can mechanically translate the old way of specifying threshold sets into activation-set-based threshold sets and retain the old semantics (and therefore the new way of specifying threshold sets generalizes the old way). In the translation, every member of the old threshold set simply becomes a singleton activation set. For example, if we wanted a *non*-short-circuiting threshold read of our AndLV under the activation-set-based semantics, our threshold set would simply be

$$\{\{(\texttt{T}, \texttt{T})\}, \{(\texttt{T}, \texttt{F})\}, \{(\texttt{F}, \texttt{T})\}, \{(\texttt{F}, \texttt{F})\}\},$$

which is a legal threshold set under the activation-set-based semantics, but has the same behavior as the old, non-short-circuiting version.

I use the activation-set-based formulation of threshold sets in Chapter 5, where I bring threshold reads to the setting of replicated, distributed data structures. I prove that activation-set-based threshold queries of distributed data structures behave deterministically (according to a definition of determinism that is particular to the distributed setting; see Section 5.3 for the details). That said, there is nothing about activation-set-based threshold sets that make them particularly suited to the distributed setting; either the original formulation of threshold sets or the even more general *threshold functions* I discuss in the following section would have worked as well.

2.6.3. Generalizing from threshold sets to threshold functions. The previous section's generalization to activation-set-based threshold sets prompts us to ask: are further generalizations possible while retaining determinism? The answer is *yes*: both the original way of specifying threshold sets and the more general, activation-set-based formulation of them can be described by *threshold functions*. A threshold function is a partial function that takes a lattice element as its argument and is undefined for all inputs that are not at or above a given element in the lattice (which I will call its *threshold point*), and *constant* for all inputs that *are* at or above its threshold point. (Note that "not at or above" is more general than "below": a threshold function is undefined for inputs that are neither above nor below its threshold point.)

Threshold functions capture the semantics of both the original style of threshold sets, and the activation-set-based style:

- In the original style of threshold sets, every element d of a threshold set can be described by a threshold function that has d as its threshold point and returns d for all inputs at or above that point.
- In the activation-set-based style of threshold sets, every element d of an activation set Q can be described by a threshold function that has d as its threshold point and returns Q for all inputs at or above that point.

In both cases, inputs for which the threshold functions are undefined correspond to situations in which the threshold read blocks.

Seen from this point of view, it becomes clear that the key insight in generalizing from the original style of threshold sets to the activation-set-based style of threshold sets is that, for inputs for which a threshold function is defined, its return value need not be its threshold point. The activation set Q is a particularly useful return value, but *any* constant return value will suffice.

LK: By itself, this is kinda unconvincing; what I should really do if I want to convince people of the determinism of threshold functions is add a generalized version of `get` based on threshold functions to

the language and prove determinism about that. IMO it ends up being less elegant than threshold sets, though, because you need many threshold functions to represent a threshold set. I think it would look something like this: the language has to be parameterized by some set F of threshold functions $f_i : D \to C$ (where C is the type of the results (which could be D, but could also be the set of all activation sets, or anything we want). Then, if someone calls `get` l $\{f_1, f_2, \dots\}$, it means that they want to get the result of threshold function f_1 if l is at or above f_1's threshold point; the result of threshold function f_2 if l is at or above f_2's threshold point, and so on. In order for this to be deterministic, we still need the notion of pairwise incompatibility: I think we need all the *threshold points* of threshold functions in a particular call to `get` to have $\top$ as their lub. Augh, maybe I should actually do this...I didn't promise I would, but maybe I should...

CHAPTER 3

Quasi-deterministic and event-driven programming with LVars

The LVars programming model presented in Chapter 2 is based on the idea of *monotonic data structures*, in which (1) information can only be added, never removed, and (2) the order in which information is added is not observable. A paradigmatic example is a set that supports insertion but not removal, but there are many others. In the LVars model, all shared data structures (called LVars) are monotonic, and the states that an LVar can take on form a *lattice*. Writes to an LVar must correspond to a least upper bound operation in the lattice, which means that they monotonically increase the information in the LVar, and that they commute with one another. But commuting writes are not enough to guarantee determinism: if a read can observe whether or not a concurrent write has happened, then it can observe differences in scheduling. So, in the LVars model, the answer to the question "has a write occurred?" (*i.e.*, is the LVar above a certain lattice value?) is always *yes*; the reading thread will block until the LVar's contents reach a desired threshold. In a monotonic data structure, the absence of information is transient—another thread could add that information at any time—but the presence of information is forever.

The LVars model guarantees determinism and supports an unlimited variety of shared data structures: anything viewable as a lattice. However, it is not as general-purpose as one might hope. Consider again the challenge problem for deterministic-by-construction parallel programming that we saw in Chapter 2:

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

In Section 2.1, we considered an attempt at a solution for this problem using purely functional parallelism (shown in Figure 2.1). Unfortunately, our purely functional attempt did not quite satisfy the above specification: although connected component discovery proceeded in parallel, members of the output set did not become available to other computations until the entire connected component had been traversed, limiting parallelism.

What would happen if we tried to solve this problem using LVars? A typical implementation of the graph-traversal part of the problem—including the purely functional implementation of Section 2.1—involves a monotonically growing set of "seen nodes"; neighbors of seen nodes are fed back into the set until it reaches a fixed point. Such fixpoint computations are ubiquitous, and would seem to be a perfect match for the LVars model due to their use of monotonicity. But they are not expressible using the threshold read and least-upper-bound write operations of the basic LVars model.

The problem is that these computations rely on *negative* information about a monotonic data structure, *i.e.*, on the *absence* of certain writes to the data structure. In a graph traversal, for example, neighboring nodes should only be explored if the current node is *not yet* in the set; a fixpoint is reached only if no new neighbors are found; and, of course, at the end of the computation it must be possible to learn exactly which nodes were reachable (which entails learning that certain nodes were not). But in the LVars model, asking whether a node is in a set means waiting until the node *is* in the set, and it is not clear how to lift this restriction while retaining determinism.

In this chapter, I describe two extensions to the basic LVars model of Chapter 2 that make it possible for monotonic data structures to "say no":

- First, I extend the model with a primitive operation `freeze` for *freezing* an LVar, which comes with the following tradeoff: once an LVar is frozen, any further writes that would change its value instead throw an exception; on the other hand, it becomes possible to discover the exact value of the LVar, learning both positive and negative information about it, without blocking.

- Second, I add the ability to attach *event handlers* to an LVar. When an event handler has been registered with an LVar, it invokes a *callback function* to run asynchronously, whenever events arrive (in the form of monotonic updates to the LVar). Ordinary LVar reads encourage a synchronous, *pull* model of programming in which threads ask specific questions of an LVar, potentially blocking until the answer is "yes". Handlers, by contrast, support an asynchronous, *push* model of programming. Crucially, it is possible to check for *quiescence* of a handler, discovering that no callbacks are currently enabled—a transient, negative property. Since quiescence means that there are no further changes to respond to, it can be used to tell that a fixpoint has been reached.

Unfortunately, freezing does not commute with writes that change an LVar.[1] If a freeze is interleaved before such a write, the write will raise an exception; if it is interleaved afterwards, the program will proceed normally. It would appear that the price of negative information is the loss of determinism!

Fortunately, the loss is not total. Although LVar programs with freezing are not guaranteed to be deterministic, they do satisfy a related property that I call *quasi-determinism*: all executions that produce a final value produce the *same* final value. To put it another way, a quasi-deterministic program can be trusted to never change its answer due to nondeterminism; at worst, it might raise an exception on some runs. This exception can in principle pinpoint the exact pair of freeze and write operations that are racing, greatly easing debugging.

In general, the ability to make exact observations of the contents of data structures is in tension with the goal of guaranteed determinism. Since pushing towards full-featured, general monotonic data structures leads to flirtation with nondeterminism, perhaps the best way of ultimately getting deterministic outcomes is to traipse a short distance into nondeterministic territory, and make our way back. The identification of quasi-deterministic programs as a useful intermediate class of programs is a contribution of this dissertation. That said, in many cases the `freeze` construct is only used as the very final

[1] The same is true for quiescence detection; see Section 3.1.2.

step of a computation: after a global barrier, freezing is used to extract an answer. In this common case, determinism is guaranteed, since no writes can subsequently occur.

I will refer to the LVars model, extended with handlers, quiescence, and freezing, as the *LVish model*. The rest of this chapter is organized as follows: in Section 3.1, I introduce the LVish programming model informally, through a series of examples. I will also return to the graph traversal I discussed above and show how to implement it using the LVish Haskell library (which I will go on to discuss the internals of in Chapter 4).

Then, in Section 3.2, I formalize the LVish model by extending the λ_{LVar} calculus of Chapter 2 to add support for handlers, quiescence, and freezing, calling the resulting language λ_{LVish}. The main technical result of this chapter is a proof of quasi-determinism for λ_{LVish} (Section 3.3). Just as with the determinism proof I gave for λ_{LVar} in Chapter 2, the quasi-determinism proof for λ_{LVish} relies on an Independence lemma (Section 3.3.4) that captures the commutative effects of LVar computations.

3.1. LVish, informally

While LVars offer a deterministic programming model that allows communication through a wide variety of data structures, they are not powerful enough to express common algorithmic patterns, like fixpoint computations, that require both positive and negative queries. In this section, I explain our extensions to the LVars model at a high level; Section 3.2 then formalizes them.

3.1.1. Asynchrony through event handlers. Our first extension to LVars is the ability to do asynchronous, event-driven programming through event handlers. An *event* for an LVar can be represented by a lattice element; the event *occurs* when the LVar's current value reaches a point at or above that lattice element. An *event handler* ties together an LVar with a callback function that is asynchronously invoked whenever some events of interest occur.

To illustrate how event handlers work, consider again the lattice of Figure 2.2(a) from Chapter 2. Suppose that *lv* is an LVar whose states correspond to this lattice. The expression

(Example 3.1) $$\texttt{addHandler}\ \mathit{lv}\ \{1, 3, 5, \dots\}\ (\lambda x.\ \texttt{put}\ \mathit{lv}\ x + 1)$$

registers a handler for *lv* that executes the callback function $\lambda x.\ \texttt{put}\ \mathit{lv}\ x + 1$ for each odd number that *lv* is at or above. When Example 3.1 is finished evaluating, *lv* will contain the smallest even number that is at or above what its original value was. For instance, if *lv* originally contains 4, the callback function will be invoked twice, once with 1 as its argument and once with 3. These calls will respectively write $1 + 1 = 2$ and $3 + 1 = 4$ into *lv*; since both writes are ≤ 4, *lv* will remain 4. On the other hand, if *lv* originally contains 5, then the callback will run three times, with 1, 3, and 5 as its respective arguments, and with the latter of these calls writing $5 + 1 = 6$ into *lv*, leaving *lv* as 6.

In general, the second argument to `addHandler`, which I call an *event set*, is an arbitrary subset Q of the LVar's lattice, specifying which events should be handled.[2] Event handlers in the LVish model are somewhat unusual in that they invoke their callback for *all* events in their event set Q that have taken place (*i.e.*, all values in Q less than or equal to the current LVar value), even if those events occurred prior to the handler being registered. To see why this semantics is necessary, consider the following, more subtle example (written in a hypothetical language with a semantics similar to that of λ_{LVar}, but with the addition of `addHandler`):

(Example 3.2)
$$\begin{aligned}\texttt{let par}\ \_ &= \texttt{put}\ \mathit{lv}\ 0\\ \_ &= \texttt{put}\ \mathit{lv}\ 1\\ \_ &= \texttt{addHandler}\ \mathit{lv}\ \{0, 1\}\ (\lambda x.\ \texttt{if}\ x = 0\ \texttt{then put}\ \mathit{lv}\ 2)\\ \texttt{in get}\ \mathit{lv}\ &\{2\}\end{aligned}$$

[2] Like threshold sets, these event sets are a mathematical modeling tool only; they have no explicit existence in the LVish library implementation.

Can Example 3.2 ever block? If a callback only executed for events that arrived after its handler was registered, or only for the largest event in its event set that had occurred, then the example would be nondeterministic: it would block, or not, depending on how the handler registration was interleaved with the `puts`. By instead executing a handler's callback once for *each and every* element in its event set below or at the LVar's value, we guarantee quasi-determinism—and, for Example 3.2, guarantee the result of 2.

The power of event handlers is most evident for lattices that model collections, such as sets. For example, if we are working with lattices of sets of natural numbers, ordered by subset inclusion, then we can write the following function:

$$\texttt{forEach} = \lambda lv.\, \lambda f.\, \texttt{addHandler}\ lv\ \{\{0\}, \{1\}, \{2\}, \dots\}\ f$$

Unlike the usual `forEach` function found in functional programming languages, this function sets up a *permanent*, asynchronous flow of data from *lv* into the callback f. Functions like `forEach` can be used to set up complex, cyclic data-flow networks, as we will see in Chapter 4.

In writing `forEach`, we consider only the singleton sets to be events of interest, which means that if the value of *lv* is some set like $\{2, 3, 5\}$ then f will be executed once for each singleton subset ($\{2\}$, $\{3\}$, $\{5\}$)—that is, once for each element. In Chapter 4, we will see that this kind of event set can be specified in a lattice-generic way, and that it corresponds closely to our implementation strategy.

3.1.2. Quiescence through handler pools. Because event handlers are asynchronous, we need a separate mechanism to determine when they have reached a *quiescent* state, *i.e.*, when all callbacks for the events that have occurred have finished running. As we saw in the introduction to this chapter, detecting quiescence is crucial for implementing fixpoint computations. To build flexible data-flow networks, it is also helpful to be able to detect quiescence of multiple handlers simultaneously. Thus, our design includes *handler pools*, which are groups of event handlers whose collective quiescence can be tested.

The simplest way to program with handler pools is to use a pattern like the following:

```
let h = newPool
  in addInPool h lv Q f;
     quiesce h
```

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

Of course, whether or not a handler is quiescent is a non-monotonic property: we can move in and out of quiescence as more `puts` to an LVar occur, and even if all states at or below the current state have been handled, there is no way to know that more `puts` will not arrive to move the LVar's state upwards in the lattice and trigger more callbacks. Early quiescence no risk to quasi-determinism, however, because `quiesce` does not yield any information about *which* events have been handled—any such questions must be asked through LVar functions like `get`. In practice, `quiesce` is almost always used together with freezing, which I explain next.

3.1.3. Freezing and the "freeze-after" pattern. Our final addition to the LVar model is the ability to *freeze* an LVar, which forbids further changes to it, but in return allows its exact value to be read. We expose freezing through the function `freeze`, which takes an LVar as its sole argument, and returns the exact value of the LVar as its result. Any `puts` that would change the value of a frozen LVar instead raise an exception, and it is the potential for races between such `puts` and `freeze` that makes the LVish model quasi-deterministic, rather than fully deterministic.

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

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

In this pattern, an event handler is registered for an LVar, subsequently quiesced, and then the LVar is frozen and its exact value is returned.

3.1.4. A parallel graph traversal using handlers, quiescence, and freezing. We can use the new features in LVish to write a parallel graph traversal in the following simple fashion:

```
traverse :: Graph → NodeLabel → Par (Set NodeLabel)
traverse g startV = do
  seen ← newEmptySet
  putInSet seen startV
  let handle node = parMapM (putInSet seen) (nbrs g node)
  freezeSetAfter seen handle
```

This code, written using the LVish Haskell library (which I will go on to describe in Chapter 4), discovers (in parallel) the set of nodes in a graph `g` reachable from a given node `startV`, and is guaranteed to produce a deterministic result.[3] It works by creating a fresh `Set` LVar (corresponding to a lattice whose elements are sets, with set union as least upper bound), and seeding it with the starting node.

The `freezeSetAfter` function implements a set-specific variant of the freeze-after pattern I described above. It combines freezing and event handlers: first, it registers the callback `handle` as a handler for the `seen` set, which will asynchronously put the neighbors of each visited node into the set, possibly triggering further callbacks, recursively. Second, when no further callbacks are ready to run—*i.e.*, when the `seen` set has reached a fixpoint—`freezeSetAfter` will freeze the set and return its exact value.

[3] In the LVish library, the `Par` type constructor is the monad in which LVar computations run; see Chapter 4 for details.

3.2. LVish, formally

In this section, I present λ_{LVish}, a core calculus for the LVish programming model. λ_{LVish} is a quasi-deterministic, parallel, call-by-value λ-calculus extended with a store containing LVars. It extends the original λ_{LVar} language of Chapter 2 to support the new features in the LVish model that I described in Section 3.1. Rather than modeling the full ensemble of event handlers, handler pools, quiescence, and freezing as separate primitives, I instead formalize the "freeze-after" pattern—which combined them—directly as a primitive. This simplifies the calculus while still capturing the essence of the programming model. I also generalize the `put` operation to allow arbitrary *update operations* (previously described in Section 2.6.1), which are inflationary and commutative, but do not necessarily compute a least upper bound.

3.2.1. Lattices. Just as with λ_{LVar}, the application-specific lattice is given as a 4-tuple $(D, \sqsubseteq, \bot, \top)$ where D is a set, $\sqsubseteq$ is a partial order on the elements of D, $\bot$ is the least element of D according to $\sqsubseteq$ and $\top$ is the greatest. The $\bot$ element represents the initial "empty" state of every LVar, while $\top$ represents the "error" state that would result from conflicting updates to an LVar. The partial order $\sqsubseteq$ represents the order in which an LVar may take on states. It induces a binary *least upper bound* (lub) operation $\sqcup$ on the elements of D. Every two elements of D must have a least upper bound in D. Formally, this makes $(D, \sqsubseteq, \bot, \top)$ a *bounded join-semilattice* with a designated greatest element $\top$; I continue to use "lattice" as shorthand, and I use D as a shorthand for the entire 4-tuple $(D, \sqsubseteq, \bot, \top)$ when its meaning is clear from the context.

3.2.2. Freezing. To model freezing, we need to generalize the notion of the state of an LVar to include information about whether it is "frozen" or not. Thus, in λ_{LVish} an LVar's *state* is a pair (d, frz), where d is an element of the application-specific set D and *frz* is a "status bit" of either true or false. A state where *frz* is false is "unfrozen", and one where *frz* is true is "frozen".

I define an ordering $\sqsubseteq_p$ on LVar states (d, frz) in terms of the application-specific ordering $\sqsubseteq$ on elements of D. Every element of D is "freezable" except $\top$. Informally:

- Two unfrozen states are ordered according to the application-specific $\sqsubseteq$; that is, $(d, \text{false}) \sqsubseteq_p (d', \text{false})$ exactly when $d \sqsubseteq d'$.
- Two frozen states do not have an order, unless they are equal: $(d, \text{true}) \sqsubseteq_p (d', \text{true})$ exactly when $d = d'$.
- An unfrozen state (d, false) is less than or equal to a frozen state (d', true) exactly when $d \sqsubseteq d'$.
- The only situation in which a frozen state is less than an unfrozen state is if the unfrozen state is $\top$; that is, $(d, \text{true}) \sqsubseteq_p (d', \text{false})$ exactly when $d' = \top$.

Adding status bits to each element (except $\top$) of the lattice $(D, \sqsubseteq, \bot, \top)$ results in a new lattice $(D_p, \sqsubseteq_p, \bot_p, \top_p)$. (The p stands for pair, since elements of this new lattice are pairs (d, frz).) I write $\sqcup_p$ for the least upper bound operation that $\sqsubseteq_p$ induces. Definitions 3.1 and 3.2 and Lemmas 3.1 and 3.2 formalize this notion.

Definition 3.1. Suppose $(D, \sqsubseteq, \bot, \top)$ is a lattice. We define an operation Freeze$(D, \sqsubseteq, \bot, \top) \triangleq (D_p, \sqsubseteq_p, \bot_p, \top_p)$ as follows:

(1) D_p is a set defined as follows:

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

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

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

(3) $\bot_p \triangleq (\bot, \text{false})$.

(4) $\top_p \triangleq (\top, \text{false})$.

Lemma 3.1 (Partition of D_p). *Suppose* $(D, \sqsubseteq, \bot, \top)$ *is a lattice, and that* $(D_p, \sqsubseteq_p, \bot_p, \top_p) = \text{Freeze}(D, \sqsubseteq, \bot, \top)$*, and let* $X = D - \{\top\}$*. Then every member of* D_p *is either*

- (d, false)*, with* $d \in D$*, or*
- (x, true)*, with* $x \in X$.

Proof. Immediate from Definition 3.1. □

Definition 3.2. We define a binary operator $\sqcup_p \in D_p \times D_p \to D_p$ as follows:

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

Lemma 3.2 says that if $(D, \leq, \bot, \top)$ is a lattice, then $(D_p, \sqsubseteq_p, \bot_p, \top_p)$ is as well:

Lemma 3.2 (Lattice structure). *Suppose that* $(D, \sqsubseteq, \bot, \top)$ *is a lattice, and that* $(D_p, \sqsubseteq_p, \bot_p, \top_p) = \text{Freeze}(D, \sqsubseteq, \bot, \top)$*. Then:*

(1) $\sqsubseteq_p$ *is a partial order over* D_p*.*

(2) *Every nonempty finite subset of* D_p *has a least upper bound.*

(3) $\bot_p$ *is the least element of* D_p*.*

(4) $\top_p$ *is the greatest element of* D_p*.*

Therefore $(D_p, \sqsubseteq_p, \bot_p, \top_p)$ *is a lattice.*

Proof. See Section A.10. □

3.2.3. Update operations. λ_{LVish} generalizes the `put` operation of λ_{LVar} to a family of operations $\texttt{put}_i$. We do so in order to allow the generalized update operations of Section 2.6.1 that are commutative and inflationary, but do not necessarily compute a least upper bound.

In order to do this, we parameterize the semantics of λ_{LVish} not only by the lattice D but also by a set U of *update operations* $u_i : D \to D$. Each u_i must meet the following conditions:

- $\forall d, i.\ \ d \sqsubseteq u_i(d)$, and
- $\forall d, i, j.\ \ u_i(u_j(d)) = u_j(u_i(d))$.

As discussed in Section 2.6.1, the first of these conditions says that each update operation is inflationary with respect to $\sqsubseteq$, and the second condition says that update operations commute with each other.

If we want to encode the original least-upper-bound semantics of `put` using $\texttt{put}_i$, we can do so by instantiating U such that there is one u_i for each element d_i of the lattice D, and defining $u_i(d)$ to be $d \sqcup d_i$. On the other hand, if D is a lattice of natural numbers and we want increment-only counters, we can instantiate U to be a singleton set $\{u\}$ where $u(d) = d + 1$. (As described in Section 2.6.1, we could also have a set of update operations $\{u_{(+1)}, u_{(+2)}, \dots\}$, where $u_{(+1)}(d)$ increments d's contents by one, $u_{(+2)}(d)$ increments by two, and so on.) Update operations are therefore general enough to express least-upper-bound writes as well as non-idempotent increments. (When a write is specifically a least-upper-bound write, I will continue to use the notation `put`, without the subscript.)

In λ_{LVish}, the `put` operation took two arguments, a location l and a lattice element d. The $\texttt{put}_i$ operations take a location l as their only argument, and $\texttt{put}_i\ l$ performs the update operation $u_i(l)$ on the contents of l.

More specifically, since l points to a state (d, frz) instead of an element d, $\texttt{put}_i\ l$ must perform u_{p_i}, a lifted version of u_i that applies to states. Each u_{p_i} takes an element of D_p as its argument and returns an element of D_p as its result. We define the set U_p of lifted operations $u_{p_i} : D_p \to D_p$ as follows:

$$
\begin{aligned}
u_{p_i}((d, \text{false})) &= (u_i(d), \text{false}) \\
u_{p_i}((d, \text{true})) &= \begin{cases} (d, \text{true}) & \text{if } u_i(d) = d \\ (\top, \text{false}) & \text{otherwise} \end{cases}
\end{aligned}
$$

3.2.4. Stores. During the evaluation of λ_{LVish} programs, a *store* S keeps track of the states of LVars. Each LVar is represented by a binding from a location l, drawn from a set *Loc*, to its state, which is some pair (d, frz) from the set D_p. The way that stores are handled in λ_{LVish} is very similar to how they are handled in λ_{LVar}, except that store bindings now point to states (d, frz), that is, elements of D_p, instead of merely to d, that is, elements of D.

Definition 3.3. A *store* is either a finite partial mapping $S : \mathit{Loc} \xrightarrow{\text{fin}} (D_p - \{\top_p\})$, or the distinguished element $\top_S$.

I use the notation $S[l \mapsto (d, \mathit{frz})]$ to denote extending S with a binding from l to (d, frz). If $l \in \mathit{dom}(S)$, then $S[l \mapsto (d, \mathit{frz})]$ denotes an update to the existing binding for l, rather than an extension. Another way to denote a store is by explicitly writing out all its bindings, using the notation $[l_1 \mapsto (d_1, \mathit{frz}_1), l_2 \mapsto (d_2, \mathit{frz}_2), \ldots]$.

We can lift the $\sqsubseteq_p$ and $\sqcup_p$ operations defined on elements of D_p to the level of stores:

Definition 3.4. A store S is *less than or equal to* a store S' (written $S \sqsubseteq_S S'$) iff:

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

Definition 3.5. The *least upper bound (lub)* of two stores S_1 and S_2 (written $S_1 \sqcup_S S_2$) is defined as follows:

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

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

If, for example,

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

then

$$[l \mapsto (d_1, \mathit{frz}_1)] \sqcup_S [l \mapsto (d_2, \mathit{frz}_2)] = \top_S.$$

Just as a store containing a binding $l \mapsto \top$ can never arise during the execution of a λ_{LVar} program, a store containing a binding $l \mapsto (\top, \mathit{frz})$ can never arise during the execution of a λ_{LVish} program. An attempted write that would take the value of l to $(\top, \text{false})$—that is, $\top_p$—will raise an error, and there is no $(\top, \text{true})$ element of D_p.

3.2.5. λ_{LVish}: syntax and semantics.

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

$$
\begin{array}{rrcl}
\text{configurations} & \sigma & ::= & \langle S;\, e\rangle \mid \textbf{error} \\
\text{expressions} & e & ::= & x \mid v \mid e\, e \mid \texttt{get}\, e\, e \mid \texttt{put}_i\, e \mid \texttt{new} \mid \texttt{freeze}\, e \\
 & & \mid & \texttt{freeze}\, e\, \texttt{after}\, e\, \texttt{with}\, e \\
 & & \mid & \texttt{freeze}\, l\, \texttt{after}\, Q\, \texttt{with}\, \lambda x.\, e, \{e, \ldots\}, H \\
\text{values} & v & ::= & \texttt{()} \mid d \mid p \mid l \mid P \mid Q \mid \lambda x.\, e \\
\text{threshold sets} & P & ::= & \{p_1,\, p_2,\, \ldots\} \\
\text{event sets} & Q & ::= & \{d_1,\, d_2,\, \ldots\} \\
\text{“handled” sets} & H & ::= & \{d_1,\, \ldots,\, d_n\} \\
\text{stores} & S & ::= & [l_1 \mapsto p_1,\, \ldots,\, l_n \mapsto p_n] \mid \top_S \\
\text{states} & p & ::= & (d, \mathit{frz}) \\
\text{status bits} & \mathit{frz} & ::= & \text{true} \mid \text{false} \\
\text{evaluation contexts} & E & ::= & [\,] \mid E\, e \mid e\, E \mid \texttt{get}\, E\, e \mid \texttt{get}\, e\, E \mid \texttt{put}_i\, E \\
 & & \mid & \texttt{freeze}\, E \mid \texttt{freeze}\, E\, \texttt{after}\, e\, \texttt{with}\, e \\
 & & \mid & \texttt{freeze}\, e\, \texttt{after}\, E\, \texttt{with}\, e \mid \texttt{freeze}\, e\, \texttt{after}\, e\, \texttt{with}\, E \\
 & & \mid & \texttt{freeze}\, v\, \texttt{after}\, v\, \texttt{with}\, v, \{e \ldots\, E\, e \ldots\}, H
\end{array}
$$

Figure 3.1. Syntax for λ_{LVish}.

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

and a set of U of update operations $u_i : D \to D$:

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

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

E-Beta

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

E-New

$$\frac{}{\langle S;\ \texttt{new}\rangle \hookrightarrow \langle S[l \mapsto (\bot, \mathsf{false})];\ l\rangle}\ (l \notin \mathit{dom}(S))$$

E-Put

$$\frac{S(l) = p_1 \qquad u_{p_i}(p_1) \neq \top_p}{\langle S;\ \texttt{put}_i\ l\rangle \hookrightarrow \langle S[l \mapsto u_{p_i}(p_1)];\ ()\rangle}$$

E-Put-Err

$$\frac{S(l) = p_1 \qquad u_{p_i}(p_1) = \top_p}{\langle S;\ \texttt{put}_i\ l\rangle \hookrightarrow \textbf{error}}$$

E-Get

$$\frac{S(l) = p_1 \qquad \mathit{incomp}(P) \qquad p_2 \in P \qquad p_2 \sqsubseteq_p p_1}{\langle S;\ \texttt{get}\ l\ P\rangle \hookrightarrow \langle S;\ p_2\rangle}$$

E-Freeze-Init

$$\frac{}{\langle S;\ \texttt{freeze}\ l\ \texttt{after}\ Q\ \texttt{with}\ \lambda x.\, e\rangle \hookrightarrow \langle S;\ \texttt{freeze}\ l\ \texttt{after}\ Q\ \texttt{with}\ \lambda x.\, e, \{\}, \{\}\rangle}$$

E-Spawn-Handler

$$\frac{S(l) = (d_1, \mathit{frz}_1) \qquad d_2 \sqsubseteq d_1 \qquad d_2 \notin H \qquad d_2 \in Q}{\langle S;\ \texttt{freeze}\ l\ \texttt{after}\ Q\ \texttt{with}\ \lambda x.\, e_0, \{e, \dots\}, H\rangle \hookrightarrow \langle S;\ \texttt{freeze}\ l\ \texttt{after}\ Q\ \texttt{with}\ \lambda x.\, e_0, \{e_0[x := d_2], e, \dots\}, \{d_2\} \cup H\rangle}$$

E-Freeze-Final

$$\frac{S(l) = (d_1, \mathit{frz}_1) \qquad \forall d_2\,.\,(d_2 \sqsubseteq d_1 \wedge d_2 \in Q \Rightarrow d_2 \in H)}{\langle S;\ \texttt{freeze}\ l\ \texttt{after}\ Q\ \texttt{with}\ v, \{v \dots\}, H\rangle \hookrightarrow \langle S[l \mapsto (d_1, \mathsf{true})];\ d_1\rangle}$$

E-Freeze-Simple

$$\frac{S(l) = (d_1, \mathit{frz}_1)}{\langle S;\ \texttt{freeze}\ l\rangle \hookrightarrow \langle S[l \mapsto (d_1, \mathsf{true})];\ d_1\rangle}$$

Figure 3.2. Reduction semantics for λ_{LVish}.

$\boxed{\sigma \longmapsto \sigma'}$

E-Eval-Ctxt

$$\frac{\langle S;\ e\rangle \hookrightarrow \langle S';\ e'\rangle}{\langle S;\ E[e]\rangle \longmapsto \langle S';\ E[e']\rangle}$$

Figure 3.3. Context semantics for λ_{LVish}.

The syntax of λ_{LVish} appears in Figure 3.1, and Figures 3.2 and 3.3 together give the operational semantics. As with λ_{LVar} in Chapter 2, both the syntax and semantics are parameterized by the lattice $(D, \sqsubseteq, \bot, \top)$, and the operational semantics is split into two parts, a *reduction semantics*, shown in Figure 2.4, and a *context semantics*, shown in Figure 2.5. The reduction semantics is also parameterized by the set U of update operations.

The λ_{LVish} grammar has most of the expression forms of λ_{LVar}: variables, values, application expressions, `get` expressions, and `new`. Instead of `put` expressions, it has $\texttt{put}_i$ expressions, which are the interface to the specified set of update operations. λ_{LVish} also adds two new language forms, the `freeze` expression and the $\texttt{freeze} - \texttt{after} - \texttt{with}$ expression, which I discuss in more detail below.

Values in λ_{LVish} include all those from λ_{LVar}—the unit value `()`, lattice elements d, locations l, threshold sets P, and λ expressions—as well as states p, which are pairs (d, frz), and event sets Q. Instead of T, I now use the metavariable P for threshold sets, in keeping with the fact that in λ_{LVish}, members of threshold sets are states p.

As before, configurations $\langle S;\ e\rangle$ are made up of a store and an expression; the *error configuration*, written **error**, is a unique element added to the set of configurations, but $\langle \top_S;\ e\rangle$ is equal to **error** for all expressions e; and the metavariable σ ranges over configurations. Both the reduction relation $\hookrightarrow$ and the context relation $\longmapsto$ are defined on configurations.

As with λ_{LVar}, the λ_{LVish} context relation $\longmapsto$ has only one rule, E-Eval-Ctxt, which allows us to apply reductions within a context. The rule itself is identical to the corresponding rule in λ_{LVar}, although the set of evaluation contexts that the metavariable E ranges over is different.

3.2.6. Semantics of `new`, `put`$_i$, and `get`. Because of the addition of status bits to the semantics, the E-New and E-Get rules have changed slightly from their counterparts in λ_{LVar}:

- `new` (implemented by the E-New rule) extends the store with a binding for a new LVar whose initial state is $(\bot, \text{false})$, and returns the location l of that LVar (*i.e.*, a pointer to the LVar).
- `get` (implemented by the E-Get rule) performs a blocking threshold read. It takes a pointer to an LVar and a *threshold set* P, which is a non-empty set of LVar states that must be *pairwise incompatible*, expressed by the premise *incomp*(P). A threshold set P is pairwise incompatible iff the lub of any two distinct elements in P is $\top_p$. If the LVar's state p_1 in the lattice is *at or above* some $p_2 \in P$, the `get` operation unblocks and returns p_2. Note that p_2 is a unique element of P, for if there is another $p_2' \neq p_2$ in the threshold set such that $p_2' \sqsubseteq_p p_1$, it would follow that $p_2 \sqcup_p p_2' = p_1 \neq \top_p$, which contradicts the requirement that P be pairwise incompatible.

λ_{LVish}'s counterpart to the `put` operation is the `put`$_i$ operation, which is actually a set of operations that provide access to the provided update operations u_i. For each u_i update operation, `put`$_i$ (implemented by the E-Put rule) takes a pointer to an LVar and updates the LVar's state to result of calling u_{p_i} on the LVar's current state, potentially pushing the state of the LVar upward in the lattice. The E-Put-Err rule applies when a `put`$_i$ operation would take the state of an LVar to $\top_p$; in that case, the semantics steps to **error**.

3.2.7. Freezing and the `freeze` $-$ `after` $-$ `with` primitive. The E-Freeze-Init, E-Spawn-Handler, E-Freeze-Final, and E-Freeze-Simple rules are all new additions to λ_{LVish}. The E-Freeze-Simple rule gives the semantics for the `freeze` expression, which takes an LVar as argument and immediately freezes and returns its contents.

More interesting is the `freeze` $-$ `after` $-$ `with` primitive, which models the "freeze-after" pattern I described in Section 3.1.3. The expression

$$\texttt{freeze}\ e_{\text{lv}}\ \texttt{after}\ e_{\text{events}}\ \texttt{with}\ e_{\text{cb}}$$

has the following semantics:

- It attaches the callback e_{cb} to the LVar e_{lv}. The expression e_{events} must evaluate to a event set Q; the callback will be executed, once, for each lattice element in Q that the LVar's state reaches or surpasses. The callback e_{cb} is a function that takes a lattice element as its argument. Its return value is ignored, so it runs solely for effect. For instance, a callback might itself do a $\texttt{put}_i$ to the LVar to which it is attached, triggering yet more callbacks.
- If the handler reaches a quiescent state, the LVar e_{lv} is frozen, and its *exact* state is returned (rather than an underapproximation of the state, as with `get`).

To keep track of the running callbacks, λ_{LVish} includes an auxiliary form,

$$\texttt{freeze}\ l\ \texttt{after}\ Q\ \texttt{with}\ \lambda x.\, e_0, \{e, \dots\}, H$$

where:

- The value l is the LVar being handled/frozen;
- The set Q (a subset of the lattice D) is the event set;
- The value $\lambda x.\, e_0$ is the callback function;
- The set of expressions $\{e, \dots\}$ are the running callbacks; and
- The set H (a subset of the lattice D) represents those values in Q for which callbacks have already been launched.

Due to λ_{LVish}'s use of evaluation contexts, any running callback can execute at any time, as if each is running in its own thread.

The rule E-Spawn-Handler launches a new callback thread any time the LVar's current value is at or above some element in Q that has not already been handled. This step can be taken nondeterministically at any time after the relevant $\texttt{put}_i$ has been performed.

The rule E-Freeze-Final detects quiescence by checking that two properties hold. First, every event of interest (lattice element in Q) that has occurred (is bounded by the current LVar state) must be handled (be in H). Second, all existing callback threads must have terminated with a value. In other words, every enabled callback has completed. When such a quiescent state is detected, E-Freeze-Final freezes the LVar's state. Like E-Spawn-Handler, the rule can fire at any time, nondeterministically, that the handler appears quiescent—a transient property! But after being frozen, any further $\texttt{put}_i$ updates that would have enabled additional callbacks will instead fault, causing the program to step to **error**.

Therefore, freezing is a way of "betting" that once a collection of callbacks have completed, no further updates that change the LVar's value will occur. For a given run of a program, either all updates to an LVar arrive before it has been frozen, in which case the value returned by $\texttt{freeze} - \texttt{after} - \texttt{with}$ is the lub of those values, or some update arrives after the LVar has been frozen, in which case the program will fault. And thus we have arrived at *quasi-determinism*: a program will always either evaluate to the same answer or it will fault.

To ensure that we will win our bet, we need to guarantee that quiescence is a *permanent* state, rather than a transient one—that is, we need to perform all updates either prior to $\texttt{freeze} - \texttt{after} - \texttt{with}$, or by the callback function within it (as will be the case for fixpoint computations). In practice, freezing is usually the very last step of an algorithm, permitting its result to be extracted. As we will see in Section 4.2.6, our LVish library provides a special `runParThenFreeze` function that does so, and thereby guarantees full determinism.

3.3. Proof of quasi-determinism for λ_{LVish}

In this section, I give a proof of quasi-determinism for λ_{LVish} that formalizes the claim made earlier in this chapter: that, for a given program, although some executions may raise exceptions, all executions that produce a final result will produce the same final result.

The quasi-determinism theorem I show says that if two executions starting from a configuration σ terminate in configurations σ' and σ'', then either σ' and σ'' are the same configuration (up to a permutation on locations), or one of them is **error**. As with the determinism proof for λ_{LVar} that I give in Section 2.5, quasi-determinism follows from a series of supporting lemmas. The basic structure of the proof follows that of the λ_{LVar} determinism proof closely. Unlike for λ_{LVar}, though, for λ_{LVish} I do not need to prove a Clash lemma (*e.g.*, Lemma 2.6) or an Error Preservation lemma (*e.g.*, Lemma 2.7). Those lemmas are unnecessary here because quasi-determinism is a weaker property than determinism.

3.3.1. Permutations and permutability. As with λ_{LVar}, the λ_{LVish} is nondeterministic with respect to the names of locations it allocates. We therefore prove quasi-determinism up to a permutation on locations. We can reuse the definition of a permutation verbatim from Section 2.5.1:

Definition 3.6 (permutation). A *permutation* is a function $\pi : \mathit{Loc} \to \mathit{Loc}$ such that:

(1) it is invertible, that is, there is an inverse function $\pi^{-1} : \mathit{Loc} \to \mathit{Loc}$ with the property that $\pi(l) = l'$ iff $\pi^{-1}(l') = l$; and

(2) it is the identity on all but finitely many elements of Loc.

We can then lift π to apply expressions, stores, and configurations. Because expressions and stores are defined slightly differently in λ_{LVish} than they are in λ_{LVar}, we must update our definitions of permutation of a store and permutation of an expression:

Definition 3.7 (permutation (expressions)). A *permutation* of an expression e is a function π defined as follows:

$$
\begin{array}{ll}
\pi(x) & = x \\
\pi(\texttt{()}) & = \texttt{()} \\
\pi(d) & = d \\
\pi(p) & = p \\
\pi(l) & = \pi(l) \\
\pi(P) & = P \\
\pi(Q) & = Q \\
\pi(\lambda x.\, e) & = \lambda x.\, \pi(e) \\
\pi(e_1\ e_2) & = \pi(e_1)\ \pi(e_2) \\
\pi(\texttt{get}\ e_1\ e_2) & = \texttt{get}\ \pi(e_1)\ \pi(e_2) \\
\pi(\texttt{put}_i\ e) & = \texttt{put}_i\ \pi(e) \\
\pi(\texttt{new}) & = \texttt{new} \\
\pi(\texttt{freeze}\ e) & = \texttt{freeze}\ \pi(e) \\
\pi(\texttt{freeze}\ e_1\ \texttt{after}\ e_2\ \texttt{with}\ e_3) & = \texttt{freeze}\ \pi(e_1)\ \texttt{after}\ \pi(e_2)\ \texttt{with}\ \pi(e_3) \\
\pi(\texttt{freeze}\ l\ \texttt{after}\ Q\ \texttt{with}\ \lambda x.\, e, \{e, \dots\}, H) & = \texttt{freeze}\ \pi(l)\ \texttt{after}\ Q\ \texttt{with}\ \lambda x.\, \pi(e), \{\pi(e), \dots\}, H
\end{array}
$$

Definition 3.8 (permutation (stores)). A *permutation* of a store S is a function π defined as follows:

$$
\begin{array}{ll}
\pi(\top_S) & = \top_S \\
\pi([l_1 \mapsto p_1, \dots, l_n \mapsto p_n]) & = [\pi(l_1) \mapsto p_1, \dots, \pi(l_n) \mapsto p_n]
\end{array}
$$

And the definition of permutation of a configuration is as it was before:

Definition 3.9 (permutation (configurations)). A *permutation* of a configuration $\langle S;\ e\rangle$ is a function π defined as follows: if $\langle S;\ e\rangle =$ **error**, then $\pi(\langle S;\ e\rangle) =$ **error**; otherwise, $\pi(\langle S;\ e\rangle) = \langle \pi(S);\ \pi(e)\rangle$.

We can now prove a Permutability lemma for λ_{LVish}, which says that a configuration σ can step to σ' exactly when $\pi(\sigma)$ can step to $\pi(\sigma')$.

Lemma 3.3 (Permutability). *For any finite permutation π,*

(1) $\sigma \hookrightarrow \sigma'$ *if and only if* $\pi(\sigma) \hookrightarrow \pi(\sigma')$.

(2) $\sigma \longmapsto \sigma'$ *if and only if* $\pi(\sigma) \longmapsto \pi(\sigma')$.

Proof. Similar to the proof of Lemma 2.1 (Permutability for λ_{LVar}); see Section A.11. TODO: Update description here once the proof is done. □

LK: For now, not saying anything about internal determinism, because I don't know if we need it.

3.3.2. Locality. Just as with the determinism proof for λ_{LVar}, proving quasi-determinism for λ_{LVish} will require us to handle expressions that can decompose into redex and context in multiple ways. An expression e such that $e = E_1[e_1] = E_2[e_2]$ can step in two different ways by the E-Eval-Ctxt rule: $\langle S;\ E_1[e_1]\rangle \longmapsto \langle S_1;\ E_1[e_1']\rangle$, and $\langle S;\ E_2[e_2]\rangle \longmapsto \langle S_2;\ E_2[e_2']\rangle$.

The Locality lemma says that $\longmapsto$ acts "locally" in each of these steps: when e_1 steps to e_1' within its context, the expression e_2 will be left alone, because it belongs to the context. Likewise, when e_2 steps to e_2' within its context, the expression e_1 will be left alone. The statement of the Locality lemma is the same as that of Lemma 2.3 (Locality for λ_{LVar}), but the proof must account for the set of possible evaluation contexts in λ_{LVish} being different (and larger).

Lemma 3.4 (Locality). *If* $\langle S;\ E_1[e_1]\rangle \longmapsto \langle S_1;\ E_1[e_1']\rangle$ *and* $\langle S;\ E_2[e_2]\rangle \longmapsto \langle S_2;\ E_2[e_2']\rangle$ *and* $E_1[e_1] = E_2[e_2]$*, then there exist evaluation contexts E_1' and E_2' such that:*

- $E_1'[e_1] = E_2[e_2']$*, and*
- $E_2'[e_2] = E_1[e_1']$*, and*
- $E_1'[e_1'] = E_2'[e_2']$.

Proof. TODO: Add short description here once the proof is done. See Section A.12. □

3.3.3. Monotonicity. The Monotonicity lemma says that, as evaluation proceeds according to the $\hookrightarrow$ relation, the store can only grow with respect to the $\sqsubseteq_S$ ordering.

Lemma 3.5 (Monotonicity). *If* $\langle S;\ e\rangle \hookrightarrow \langle S';\ e'\rangle$*, then* $S \sqsubseteq_S S'$.

Proof. Straightforward by cases on the rule of the reduction semantics by which $\langle S;\ e\rangle$ steps to $\langle S';\ e'\rangle$. The interesting cases are for the E-New and E-Put rules. See Section A.13. TODO: Update description here once the proof is done. Freezing might be interesting, too! □

3.3.4. Independence. Like the lemma for λ_{LVar} of the same name (Lemma 2.5), the Independence lemma establishes a "frame property" for λ_{LVish} that captures the idea that independent effects commute with each other. Just as for λ_{LVar}, Lemma 3.6 requires as a precondidion that the store S'' must be *non-conflicting* (Definition 2.10) with the original transition from $\langle S;\ e\rangle$ to $\langle S';\ e'\rangle$, meaning that locations in S'' cannot share names with locations newly allocated during the transition, which rules out location name conflicts caused by allocation.

In the context of λ_{LVish}, where freezing is possible, we have an additional precondition that the stores $S' \sqcup_S S''$ and S are *equal in status*—that, for all the locations shared between them, the status bits of those locations agree. This assumption rules out interference from freezing.

Definition 3.10. Two stores S and S' are *equal in status* (written $S =_{\mathit{frz}} S'$) iff for all $l \in (\mathit{dom}(S) \cap \mathit{dom}(S'))$,
if $S(l) = (d, \mathit{frz})$ and $S'(l) = (d', \mathit{frz}')$, then $\mathit{frz} = \mathit{frz}'$.

Lemma 3.6 (Independence). *If* $\langle S;\ e\rangle \hookrightarrow \langle S';\ e'\rangle$ *(where* $\langle S';\ e'\rangle \neq$ **error***), then we have that:*

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

where S'' *is any store meeting the following conditions:*

- S'' *is non-conflicting with* $\langle S;\ e\rangle \hookrightarrow \langle S';\ e'\rangle$,
- $S' \sqcup_S S'' =_{frz} S$, *and*
- $S' \sqcup_S S'' \neq \top_S$.

Proof. By cases on the rule of the reduction semantics by which $\langle S;\ e\rangle$ steps to $\langle S';\ e'\rangle$. The interesting cases are for the E-New and E-Put rules. Since $\langle S';\ e'\rangle \neq$ **error**, we do not need to consider the E-Put-Err rule. See Section A.14. □

3.3.5. Quasi-Confluence. Lemma 3.7 The Strong Local Quasi-Confluence lemma says that if a configuration σ can step to configurations σ_a and σ_b, then one of two possibilities is true: either there exists a configuraton σ_c that σ_a and σ_b can each reach in at most one step, modulo a permutation on locations, *or* at least one of σ_a or σ_b steps to **error**. Lemmas 3.8 and 3.9 then generalize that result to arbitrary numbers of steps.

Lemma 3.7 (Strong Local Quasi-Confluence). *If* $\sigma \longmapsto \sigma_a$ *and* $\sigma \longmapsto \sigma_b$*, then either:*

(1) *there exist* σ_c, i, j, π *such that* $\sigma_a \longmapsto^i \sigma_c$ *and* $\pi(\sigma_b) \longmapsto^j \sigma_c$ *and* $i \leq 1$ *and* $j \leq 1$*, or*

(2) $\sigma_a \longmapsto$ **error** *or* $\sigma_b \longmapsto$ **error**.

Proof. Similar to the proof of Lemma 2.8 (Strong Local Confluence for λ_{LVar}); see Section A.15. TODO: Update description here once the proof is done. □

Lemma 3.8 (Strong One-Sided Quasi-Confluence). *If* $\sigma \longmapsto \sigma'$ *and* $\sigma \longmapsto^m \sigma''$*, where* $1 \leq m$*, then either:*

(1) *there exist* σ_c, i, j, π *such that* $\sigma' \longmapsto^i \sigma_c$ *and* $\pi(\sigma'') \longmapsto^j \sigma_c$ *and* $i \leq m$ *and* $j \leq 1$*, or*

(2) *there exists* $k \leq m$ *such that* $\sigma' \longmapsto^k$ **error***, or there exists* $k \leq 1$ *such that* $\sigma'' \longmapsto^k$ **error**.

Proof. By induction on m; see Section A.16. □

Lemma 3.9 (Strong Quasi-Confluence). *If* $\sigma \longmapsto^n \sigma'$ *and* $\sigma \longmapsto^m \sigma''$*, where* $1 \leq n$ *and* $1 \leq m$*, then either:*

(1) *there exist* σ_c, i, j, π *such that* $\sigma' \longmapsto^i \sigma_c$ *and* $\pi(\sigma'') \longmapsto^j \sigma_c$ *and* $i \leq m$ *and* $j \leq n$*, or*

(2) *there exists* $k \leq m$ *such that* $\sigma' \longmapsto^k$ **error***, or there exists* $k \leq n$ *such that* $\sigma'' \longmapsto^k$ **error***.*

Proof. By induction on n; see Section A.17. □

Lemma 3.10 (Quasi-Confluence). *If* $\sigma \longmapsto^* \sigma'$ *and* $\sigma \longmapsto^* \sigma''$*, then either:*

(1) *there exist* σ_c *and* π *such that* $\sigma' \longmapsto^* \sigma_c$ *and* $\pi(\sigma'') \longmapsto^* \sigma_c$*, or*

(2) $\sigma' =$ **error** *or* $\sigma'' =$ **error***.*

Proof. Strong Quasi-Confluence (Lemma 3.9) implies Quasi-Confluence. □

3.3.6. Quasi-determinism theorem. The Quasi-Determinism theorem, Theorem 3.1, is a straightforward result of Lemma 3.10. It says that if two executions starting from a configuration σ terminate in configurations σ' and σ'', then σ' and σ'' are the same configuration, or one of them is **error**.

Theorem 3.1 (Quasi-Determinism). *If* $\sigma \longmapsto^* \sigma'$ *and* $\sigma \longmapsto^* \sigma''$*, and neither* σ' *nor* σ'' *can take a step, then either:*

(1) *there exists* π *such that* $\sigma' = \pi(\sigma'')$*, or*

(2) $\sigma' =$ **error** *or* $\sigma'' =$ **error***.*

Proof. By Lemma 3.10, one of the following two cases applies:

(1) There exists σ_c and π such that $\sigma' \longmapsto^* \sigma_c$ and $\pi(\sigma'') \longmapsto^* \sigma_c$. Since σ' cannot step, we must have $\sigma' = \sigma_c$.

By Lemma 3.3 (Permutability), σ'' can step iff $\pi(\sigma'')$ can step, so since σ'' cannot step, $\pi(\sigma'')$ cannot step either.

Hence we must have $\pi(\sigma'') = \sigma_c$. Since $\sigma' = \sigma_c$ and $\pi(\sigma'') = \sigma_c$, $\sigma' = \pi(\sigma'')$.

(2) $\sigma' =$ **error** or $\sigma'' =$ **error**, and so the result is immediate.

□

3.3.7. Discussion: quasi-determinism in practice. LK: I kinda threw this subsection in here on a whim. Maybe it should actually go somewhere in Chapter 4, or maybe it should be its own section.

The quasi-determinism result for λ_{LVish} shows that it is not possible to get multiple "answers" from the same program: every run will either produce the same answer or an error. Importantly, this property is true not only for programs that use the freeze-after pattern expressed by the `freeze − after − with` primitive, but even those that freeze in arbitrary places using the simpler `freeze` primitive. This means that in practice, in a programming model based on LVars with freezing and handlers, even a program that fails to ensure quiescence (introducing the possiblility of a race between a `put` and a `freeze`) cannot produce multiple non-**error** answers.

Therefore the LVish programming model is fundamentally different from one in which the programmer must manually insert synchronization barriers to prevent data races. In that kind of a model, a program with a misplaced synchronization barrier can be fully nondeterministic, producing multiple observable answers. In the LVish model, the worst that can happen is that the program raises an error. Moreover, in the LVish model, an **error** result *always* means that there is an undersynchronization bug in the program, and in principle the error message can even specify exactly which write operation happened after which freeze operation, making it easier to debug the problem.

However, if we *can* ensure that an LVar is only ever frozen *after* all writes to that LVar have completed, then we can guarantee full determinism, because we will have ruled out races between write operations and freeze operations. In the next chapter, I discuss how the LVish Haskell library enforces this "freeze after writing" property.

CHAPTER 4

The LVish library: interface, implementation, and evaluation

We want the programming model of Chapters 2 and 3 to be realizable in practice. If the determinism guarantee offered by LVars is to do us any good, however, we need to add LVars to a programming environment that is already deterministic.

The *monad-par* Haskell library [35], which provides the `Par` monad, is one such deterministic parallel programming environment. Haskell is in general an appealing substrate for implementing guaranteed-deterministic parallel programming models because it is pure by default, and its type system enforces separation of pure and effectful code via monads. In order for the determinism guarantee of any parallel programming model to hold, the only side effects allowed must be those sanctioned by the programming model.[1] In the case of the basic LVars model, those effects are `put` and `get` operations on LVars. LK: Should we say that both `put` and `get` are effects, or just `put`? Implementing the LVars model as a Haskell library makes it possible to provide compile-time guarantees about determinism and quasi-determinism, because programs written using the library run in a particular monad and can therefore only perform the side effects that we choose to sanction.

Another reason why the existing `Par` monad is an appealing starting point for a practical implementation of LVars is that it allows inter-task communication through IVars, which, as we have seen, are a special case of LVars. Implementing the basic LVars model then becomes a matter of generalizing the existing `Par` monad implementation. Finally, the `Par` monad approach is appealing because it is implemented entirely as a library in Haskell, with a library-level scheduler. This modularity makes it

[1] Haskell is often advertised as a purely functional programming language, that is, one without side effects, but it is perhaps more useful to think of it as a language that gets other effects out of the way so that one can add one's own effects!

possible to make changes to the `Par` scheduling strategy (which we will need to do in order to support LVars) without having to make any modifications to GHC or its run-time system.

LK: Ugh, I hate roadmap paragraphs, and this is just about the worst of all time. Maybe I can make this better.

In this chapter, I describe the *LVish* library, a Haskell library for practical deterministic and quasi-deterministic parallel programming with LVars. We have already had a taste of what it is like to use the LVish library; Section 3.1 gave an example of an LVish Haskell program. Here, I go on to give a systematic introduction to the LVish library. First, in Section 4.1, I discuss the library at a high level. Then, in Section 4.2, I describe the LVish API, and I discuss the parallel breadth-first traversal of Section 3.1 in more detail. In Section 4.3, I discuss how LVish supports *imperative disjoint parallelism* (in the style of DPJ [**8**]) alongside LVars, which we accomplish using a monad transformer, `ParST`. In Section 4.4, I describe how the LVish library itself is implemented. Finally, in Sections 4.5 and 4.6, I present two case studies of parallelizing existing Haskell programs by porting them to LVish. First, in Section 4.5, I describe using LVish to parallelize a control-flow analysis (k-CFA) algorithm. Second, in Section 4.6, I describe using LVish to parallelize *PhyBin*, a bioinformatics application that relies heavily on a (parallelizable) tree-edit distance algorithm.

4.1. The big picture

Our library adopts and builds on the basic approach of the `Par` monad and the monad-par library [**35**], enabling us to employ our own notion of lightweight, library-level threads with a custom scheduler. It supports the programming model laid out in Section 3.1 in full, including explicit handler pools. It differs from the formalism of Section 3.2 in following Haskell's by-need evaluation strategy, which also means that concurrency in the library is *explicitly marked*, either through uses of a `fork` function or through asynchronous callbacks, which run in their own lightweight thread.

We envision two parties interacting with the LVish library. First, there are *data structure authors*, who use the library directly to implement a specific monotonic data structure (*e.g.*, a monotonically growing finite map). Second, there are *application writers*, who are clients of these data structures. Only the application writers receive a (quasi-)determinism guarantee; an author of a data structure is responsible for ensuring that the states their data structure can take on correspond to the elements of a lattice, and that the exposed interface to it corresponds to some use of `put`, `get`, `freeze`, and event handlers.

Thus, our library is focused primarily on *lattice-generic* infrastructure: the `Par` monad itself, a thread scheduler, support for blocking and signaling threads, handler pools, and event handlers. Since this infrastructure is unsafe—that is, it does not guarantee determinism or quasi-determinism—only data structure authors should import it, subsequently exporting a *limited* interface specific to their data structure. For finite maps, for instance, this interface might include key/value insertion, lookup, event handlers and pools, and freezing—along with higher-level abstractions built on top of these.

For this approach to scale well with available parallel resources, it is essential that the data structures themselves support efficient parallel access; a finite map that was simply protected by a global lock would force all parallel threads to sequentialize their access. Thus, we expect data structure authors to draw from the extensive literature on scalable parallel data structures, employing techniques like fine-grained locking and lock-free data structures [28]. Data structures that fit into the LVish model have a special advantage: because all updates must commute, it may be possible to avoid the expensive synchronization which *must* be used for non-commutative operations [4]. And in any case, monotonic data structures are usually much simpler to represent and implement than general ones. LK: I think Aaron wrote that last sentence. I'm not sure I feel comfortable making the claim without citation...

4.2. The LVish library interface for application writers

In this section I illustrate the use of the LVish library from the point of view of the application writer, through a series of examples.[2]

4.2.1. Basic examples using IVars. As mentioned in the previous section, the LVish library extends the approach of Haskell's monad-par library for deterministic parallelism, which allows communication between parallel tasks through IVars. Recall that IVars can only be assigned to once. Therefore the following program written using `Par` will deterministically raise an error, because it tries to write to the IVar `num` twice:

```
import Control.Monad.Par

p :: Par Int
p = do
  num ← new
  fork (put num 3)
  fork (put num 4)
  get num

main = print (runPar p)
```

Here, `p` is a computation of type `Par Int`, meaning that it runs in the `Par` monad (via the call to `runPar`) and returns a value of `Int` type. `num` is an IVar, created with a call to `new` and then assigned to via two calls to `put`, each of which runs in a separately `fork`ed task. The `runPar` function is an implicit global barrier: all `fork`s have to complete before `runPar` can return.

The program raises a "multiple put" error at runtime, which is as it should be: differing writes to the same shared location could cause the subsequent call to `get` to behave nondeterministically. (Here,

[2]The examples in this section, among others, are available at https://github.com/lkuper/lvar-examples/. All of the examples are tested against GHC 7.6.3, the most recent release at the time of this writing.TODO: I should test against 7.8!

`get` has IVar semantics, not LVar semantics: rather than performing a threshold read, it blocks until `num` has been written, then unblocks and evaluates to the exact contents of `num`.)

However, in monad-par, even multiple writes of the *same* value to an IVar will raise a "multiple put" error:

```
import Control.Monad.Par

p :: Par Int
p = do
  num ← new
  fork (put num 4)
  fork (put num 4)
  get num

main = print (runPar p)
```

This program differs from the previous one only in that the two `put`s are writing 4 and 4, rather than 3 and 4. Even though the call to `get` would produce a deterministic result regardless of which write happened first, the program still raises an error because of the IVar single-write restriction.

Now let's consider a version of this program written using the LVish library. (Of course, in LVish we are not limited to IVars, but we will consider IVars first as an interesting special case of LVars, and then go on to consider some more sophisticated LVars later in this section.) A version of the above program written using LVish will write 4 to an IVar twice and then deterministically print 4 instead of raising an error:

```
{-# LANGUAGE DataKinds #-}
{-# LANGUAGE TypeFamilies #-}

import Control.LVish  -- Generic scheduler; works with all LVars.
import Data.LVar.IVar -- The particular LVar we need for this program.

p :: (HasPut e, HasGet e) ⇒ Par e s Int
p = do
```

```
  num ← new
  fork (put num 4)
  fork (put num 4)
  get num

main = print (runPar p)
```

Here, we import the `Control.LVish` module rather than `Control.Monad.Par` (that is, we are using LVish instead of monad-par), and we must specifically import `Data.LVar.IVar` in order to specify which LVar data structure we want to work with (since we are no longer limited to IVars). Just as with monad-par, the LVish `runPar` function is a global barrier: both `fork`s must compete before `runPar` can return. Also, as before, we have `new`, `put`, and `get` operations that respectively create, update, and read from `num`. However, these operations now have LVar semantics: the `put` operation performs a least-upper-bound write, and the `get` operation performs a threshold read, where the threshold set is implicitly the set of all `Int`s. We do not need to explicitly write down the threshold set in the code; rather, it is the obligation of the `Data.LVar.IVar` module to provide operations (`put` and `get`) that have the semantic effect of least-upper-bound writes and threshold reads (as I touched on earlier in Section 2.2.3).

There are two other important differences between the monad-par program and the LVish program: the `Par` type constructor has gained two new type parameters, `e` and `s`, and `p`'s type annotation now has a *type class constraint* of `(HasPut e, HasGet e)`. Furthermore, we have added two `LANGUAGE` pragmas, instructing the compiler that we are now using the `DataKinds` and `TypeFamilies` language extensions. In the following section, I explain these changes.

4.2.2. The `e` and `s` type parameters: effect tracking and session tracking. In order to support both determinism and quasi-determinism guarantees in the LVish library, we need to be able to guarantee that only certain LVar effects can occur within a given `Par` computation. In a deterministic computation, only update operations (such as `put`) and threshold reads should be allowed; in a quasi-deterministic computation, `freeze` operations should be allowed as well. Yet other combinations may be desirable:

for instance, we may want a computation to perform *only* writes, and not reads. Furthermore, we want to be able to specifically allow or disallow *non-idempotent* update operations, which I discuss in more detail in Section 4.4.6.

In order to capture these constraints and make them explicit in the types of LVar computations, LVish indexes `Par` computations with a *phantom type* `e` that indicates their *effect level*. The `Par` type becomes, instead, `Par e`, where `e` is a type-level encoding of booleans indicating which operations, such as writes, reads, or freeze operations, are allowed to occur inside it.

LK: Any terminology preference for "effect level" vs. "effect signature"?

Since we do not know in advance which effects this "switch-on-and-off" capability will need to support, LVish follows the precedent of Kiselyov *et al*. on extensible effects in Haskell [30]: it abstracts away the specific structure of `e` into *type class constraints*, which allow a `Par` computation to be annotated with the *interface* that its `e` type parameter is expected to satisfy. This static effect tracking mechanism allows us to define "effect shorthands" and use them as Haskell type class constraints. For example, a `Par` computation where `e` is annotated with the effect level constraint `HasPut` can perform `puts`. In our example above, `e` is annotated with both `HasPut` and `HasGet` and therefore the `Par` computation in question can perform both `puts` and `gets`. We will see several more examples of effect level constraints in LVish `Par` computations shortly.

The effect tracking infrastructure is also the reason why we need to use the `DataKinds` and `TypeFamilies` language extensions in our LVish programs. For brevity, I will elide the `LANGUAGE` pragmas in the rest of the example LVish programs in this section. TODO: I need to figure out and actually explain *why* the effect tracking requires `DataKinds` and `TypeFamilies`.

The LVish `Par` type constructor also has a second type parameter, `s`, making `Par e s a` the complete type of a `Par` computation that returns a result of type `a`. The `s` parameter ensures that, when a computation in the Par monad is run using the provided `runPar` operation (or using a variant of `runPar`, which I will discuss below), it is not possible to return an LVar from `runPar` and reuse it in another

call to `runPar`. The `s` type parameter also appears in the types of LVars themselves, and the universal quantification of `s` in `runPar` and its variants forces each LVar to be tied to a single "session", *i.e.*, a single use of a `run` function, in the same way that the ST monad in Haskell prevents an `STRef` from escaping `runST`. Doing so allows the Lvish implementation to assume that LVars are created and used within the same session.[3]

4.2.3. Non-idempotent writes: an increment-only counter. TODO: I want to put an example here, but the last time I tried actually using `Data.LVar.Counter` in LVish, it didn't work. :(Need to figure this out!

4.2.4. Container LVars: a shopping cart example. For our next few examples, let us consider concurrently adding items to a shopping cart. Suppose we have an `Item` data type for items that can be added to the cart. For the sake of this example, suppose that only two items are on offer:

```
data Item = Book | Shoes
  deriving (Show, Ord, Eq)
```

The cart itself can be represented using the `IMap` LVar type (provided by the `Data.LVar.PureMap` module), which is a key-value map where the keys are `Items` and the values are the quantities of each item.[4]

LK: It's probably a little late to be complaining about this, but I really don't like the names 'IMap and ISet. They're by analogy with IVar, but the "i" originally stood for "immutable", and they're *not* immutable! Any way we can change them? Maybe to LMap, LSet, etc.?

[3]The addition of the `s` type parameter to `Par` in the LVish library has nothing to do with LVars in particular; it would also be a useful addition to the original `Par` library to prevent programmers from reusing an IVar from one `Par` computation to another, which is, as Simon Marlow has noted, "a Very Bad Idea; don't do it" [33].

[4]The "Pure" in `Data.LVar.PureMap` distinguishes it from LVish's other map data structure, which is also called `IMap`, but is provided by the `Data.LVar.SLMap` module and is a lock-free data structure based on concurrent skip lists. The `IMap` provided by `Data.LVar.PureMap`, on the other hand, is a reference implementation of a map, which uses a pure `Data.Map` wrapped in a mutable container. Both `IMaps` present the same API, but the lock-free version is designed to scale as parallel resources are added. I discuss the role of lock-free data structures in LVish in more detail in Section 4.5.5; in any case, either implementation of `IMap` would have worked for this example.

```
import Control.LVish
import Data.LVar.PureMap

p :: (HasPut e, HasGet e) ⇒ Par e s Int
p = do
  cart ← newEmptyMap
  fork (insert Book 2 cart)
  fork (insert Shoes 1 cart)
  getKey Book cart

main = print (runPar p)
```

Here, the `newEmptyMap` operation creates a new `IMap`, and the `insert` operation allows us to add new key-value pairs to the cart. In this case, we are concurrently adding the `Book` item with a quantity of 2, and the `Shoes` item with a quantity of 1.

The `getKey` operation allows us to threshold on a key, in this case `Book`, and get back the value associated with that key once it has been written. The (implicit) threshold set of a call to `getKey` is the set of all values that might be associated with a key; in this case, the set of all `Int`s. This is a legal threshold set because in this example, map entries are *immutable*: we cannot, for instance, insert a key of `Book` with a quantity of 2 and then later change the 2 to 3. In a more realistic shopping cart, the values in the cart could themselves be LVars representing incrementable counters, as in the previous section. TODO: I'd like there to be some kind of footnote here about the problem that LVars-that-contain-LVars presents for determinism. I don't actually understand the problem, though. However, a shopping cart from which we can *delete* items is not possible because it would go against the principle of monotonic growth.[5]

[5] On the other hand, one way to encode a container that allows both insertion and removal of elements is to represent it internally with *two* containers, one for the inserted elements and one for the removed elements, where both containers grow monotonically. *Conflict-free replicated data types* (CRDTs) [45] use variations on this approach to encode counters that support decrements as well as increments, sets that support removals as well as additions, and other data structures that support seemingly non-monotonic operations. I discuss the relationship of LVars to CRDTs in more detail in Chapter 5.

In this program, the call to `getKey` will be able to unblock as soon as the first `insert` operation has completed, and the program will deterministically print 2 regardless of whether the second `insert` has completed at the time that `getKey` unblocks.

4.2.5. A quasi-deterministic shopping cart example. So far, the examples in this section have been fully deterministic; they do not use `freeze`. Next, let us consider a program that freezes and reads the exact contents of a shopping cart, concurrently with inserting into it.

```
import Control.LVish
import Control.LVish.DeepFrz
import Data.LVar.PureMap
import qualified Data.Map as M

p :: (HasPut e, HasFreeze e) ⇒ Par e s (M.Map Item Int)
p = do
  cart ← newEmptyMap
  fork (insert Book 2 cart)
  fork (insert Shoes 1 cart)
  freezeMap cart

main = do
  v ← runParQuasiDet p
  print (M.toList v)
```

Here, we are `insert`ing items into our cart and then calling `getKey` on the `Book` key, as before. But, instead of returning the result of the call to `getKey`, this time `p` returns the result of a call to `freezeMap`. Note that the return type of `p` is a `Par` computation containing not an `Int`, but rather an entire map from `Item`s to `Int`s.

In fact, this map is not the `IMap` that `Data.LVar.PureMap` provides, but rather the standard `Map` from the `Data.Map` module (imported as `M`). This is possible because `Data.LVar.PureMap` is implemented using `Data.Map`, and so freezing its `IMap` simply returns the underlying `Data.Map`.

Because p performs a freezing operation, the effect level of its return type must reflect the fact that it is allowed to perform freezes. Therefore, in addition to `HasPut`, we now have the additional type class constraint of `HasFreeze` on `e`. Furthermore, because p is allowed to perform a freeze, we cannot run it with `runPar`, as in our previous examples, but must instead use a special variant, `runParQuasiDet`, whose type signature allows `Par` computations that allow freezing to be passed to it.

The quasi-determinism in this program arises from the fact that the call to `freezeMap` may run before both `fork`ed computations have completed. In this example, one or both calls to `insert` may run after the call to `freezeMap`. If this happens, the program will raise a write-after-freeze exception. The other possibility is that both items are already in the cart at the time it is frozen, in which case the program will run without error and print both items. There are therefore two possible outcomes: a cart with both items, or a write-after-freeze error. The advantage of quasi-determinism is that it is not possible to get multiple *non-error* outcomes, such as, for instance, an empty cart or a cart in which only the `Book` has been written.

4.2.6. Regaining full determinism with `runParThenFreeze`. The advantage of freezing is that it allows us to observe the exact, complete contents of an LVar; the disadvantage is that it introduces quasi-determinism due to the possibility of a write racing with a freeze, as in the example above. But, if we could ensure that a `freeze` operation happened *last*, we would be able to freeze LVars with no risk to determinism. In fact, the LVish library offers a straightforward solution to this problem: instead of writing `freeze` operations ourselves (and perhaps accidentally writing an undersynchronized program that freezes an LVar too early), we can ask LVish to freeze an LVar for us itself while "on the way out" of a `Par` computation. The mechanism that allows this is a special `runParThenFreeze` function. A version of the above program written using `runParThenFreeze` is as follows:

```
import Control.LVish
import Control.LVish.DeepFrz
import Data.LVar.PureMap
```

```
p :: (HasPut e) ⇒ Par e s (IMap Item s Int)
p = do
  cart ← newEmptyMap
  fork (insert Book 2 cart)
  fork (insert Shoes 1 cart)
  return cart

main = print (runParThenFreeze p)
```

An interesting thing to note about this `Par` computation is that it *only* performs writes (as evidenced by its effect level, which is only constrained by `HasPut`). Also, unlike the previous version, where the `freeze` took place inside the `Par` computation, this computation returns an `IMap` rather than a `Map`.

Because there is no synchronization operation after the two `fork` calls, `p` may well return `cart` before both (or either) of the items have been added to it. However, since `runParThenFreeze` is an implicit global barrier (just as `runPar` and `runParQuasiDet` are), both calls to `insert` *must* complete before `runParThenFreeze` can return—which means that the result of the program is deterministic.

4.2.7. Event-driven programming: a deterministic parallel graph traversal. Finally, we'll look at an example that uses event handlers as well as freezing. The function `traverse` takes a graph `g` and a vertex `startNode` and finds the set of all vertices reachable from `startNode`, in parallel.

```
import Control.LVish
import Control.LVish.DeepFrz
import Data.LVar.Generic (addHandler, freeze)
import Data.LVar.PureSet

traverse :: (HasPut e, HasGet e, HasFreeze e) ⇒ G.Graph → Int → Par e s (ISet Frzn Int)
traverse g startNode = do
  seen ← newEmptySet
  h ← newHandler seen
       (λnode → do
           mapM (λv → insert v seen)
             (neighbors g node)
           return ())
  insert startNode seen -- Kick things off
```

```
  quiesce h
  freeze seen

main = do
  v ← runParQuasiDet (traverse myGraph (0 :: G.Vertex))
  print (fromISet v)
```

`traverse` works by first creating a new LVar of set type, called `seen`. Its next step is to attach an event handler to `seen`, using the `newHandler` function. `newHandler` takes two arguments: an LVar and the callback that we want to run every time an event occurs on that LVar (in this case, every time a new node is added to the set).[6] We respond to such events by looking up the neighbors of the newly arrived node (with a call to the `neighbors` operation, which takes a graph and a vertex and returns a list of the vertex's neighbor vertices), then mapping the `insert` function over that list of neighbors.

Finally, `traverse` adds the starting node to the `seen` set by calling `insert startNode seen`—and the event handler does the rest of the work. We know that we are done handling events when the call to `quiesce h` returns; it will block until all events have been handled. Finally, we freeze and return the LVar, which by this point is a set of all reachable nodes.

The good news is that this particular graph traversal is deterministic. The bad news is that, in general, freezing introduces quasi-determinism, since we could have accidentally forgotten to call `quiesce` before the freeze—which is why `traverse` must be run with `runParQuasiDet`, rather than `runPar`. Although the program is deterministic, the *language-level* guarantee is merely of quasi-determinism, not determinism.

However, just as with the final shopping cart example above, we can use `runParThenFreeze` to ensure that freezing happens last. Here is a version of `traverse` that is guaranteed to be deterministic at the language level:

```
import Control.LVish
```

[6] `newHandler` is not provided by LVish, but we can easily implement it using LVish's built-in `newPool` and `addHandler` operations.LK: Actually, is there any good reason why LVish *doesn't* provide something like `newHandler`?

```
import Control.LVish.DeepFrz
import Data.LVar.Generic (addHandler, freeze)
import Data.LVar.PureSet

traverse :: HasPut e ⇒ G.Graph → Int → Par e s (ISet s Int)
traverse g startNode = do
  seen ← newEmptySet
  h ← newHandler seen
       (λnode → do
           mapM (λv → insert v seen)
             (neighbors g node)
           return ())
  insert startNode seen -- Kick things off
  return seen

main = print (runParThenFreeze (traverse myGraph (0 :: G.Vertex)))
```

Here, since freezing is performed by `runParThenFreeze` rather by an explicit call to `freeze`, it is no longer necessary to explicitly call `quiesce`, either! The reason for this is that the implicit barrier of `runParThenFreeze` will ensure that all outstanding events that can be handled will be handled before it can return.

4.3. `Par`-monad transformers and disjoint parallel update

LK: This section is from sections 4 and 5 of the PLDI paper. I'm generally a bit unhappy with it—it's too Haskelly compared to the surrounding material, and there's probably more detail than there needs to be; I'd like to pare it down to the bare minimum needed to explain `ParST` (can we get away with not explaining monad transformers?). Also, the PLDI paper had an argument here as to why `ParST` retains determinism, which I've left out because it's kind of handwavy and unconvincing.

LK: Really, all we want to say is this: We can apply something called a ParST transformer to the Par monad that will let us thread some state through, and at a fork the state has to be either split or duplicated. If we're splitting, say, a vector of locations, the Haskell type system is powerful enough to ensure at compile time that neither of the child computations can access the original complete vector.

The effect-tracking system of the previous section provides a way to toggle on and off a fixed set of basic capabilities using the type system—that is, with the switches embedded in the `e` effect signature that parameterizes the `Par` type. These type-level distinctions are needed for defining restricted but safe idioms, but they do not address *extensibility*. For that, we turn to multiple monads rather than a single parameterized `Par` monad.

TODO: This "extensibility" point is kind of out of place here—it made sense in the context of the PLDI paper, but here it's not clear what I'm talking about.

4.3.1. Monad transformers and their use in LVish. Haskell programmers use a variety of different monads: `Reader` for threading parameters, `State` for in-place update, `Cont` for continuations, and so on. All monads support the same core operations (`bind` and `return` from the `Monad` type class) and satisfy the three monad laws. However, each monad must also provide other operations that make it worth using. Most famously, the `IO` monad provides various input-output operations.

A monad *transformer*, on the other hand, is a type constructor that adds "plug-in" capabilities to an underlying monad. For example, the `StateT` monad transformer adds an extra piece of implicit, modifiable state to an underlying monad. Adding a monad transformer to a type always returns another monad (preserving the `Monad` instance).

In the same way, we can define a *`Par`-monad transformer* as a type constructor `T`, where, for all `Par` monads `m`, `T m` is another `Par` monad with additional capabilities, and where a value of type `T m a`, for instance, `T (Par e s) a`, is a computation in that monad. Indeed, `Par`-monad transformers *are* valid monad transformers (in the sense of providing a standard `MonadTrans` instance).

The semantics of a Par monad is captured by a series of type classes, all of which are closed under Par-monad transformer application. At minimum, an instance of the Par monad type class must provide the fork operation that we have seen in our examples so far:

```
class (Monad m) ⇒ ParMonad m where
  fork :: m () → m ()
```

Programs with fork create a binary tree of monadic actions with () (unit) return values. Additional type classes capture the interfaces to basic parallel data structures and control constructs such as futures (ParFuture), IVars (ParIVar), and more general LVars (ParLVar). For example, the class ParIVar provides new and put operations with the signatures below.[7]

```
class (ParMonad m) ⇒ ParFuture m  where
  ...
class (ParMonad m) ⇒ ParIVar m  where
  type IVar m :: * → *
  new :: m (IVar m a)
  put :: IVar m a → a → m ()
  get :: IVar m a → m a
```

The ParFuture, ParIVar, and ParLVar type classes form a hierarchy: any implementation that can support LVars can support IVars, and any that can support IVars can support futures. Taken together, this framework for generic Par programming makes it possible for LVish programs to be reusable across a variety of schedulers. This can be quite useful; for example, LVish provides a ParFuture instance for the native GHC work-stealing scheduler [36].

4.3.2. Example: threading state in parallel. Perhaps the simplest example of a Par-monad transformer is the standard StateT monad transformer (provided by Haskell's Control.Monad.State package). However, even if m is a Par monad, for StateT s m to also be a Par monad, the state s

[7] Although it may appear that generic treatment of Par monads as type variables m removes the additional metadata in a type such as Par e s a, note that it is possible to recover this information with type-level functions.

must be *splittable*; that is, it must be specified what is to be done with the state at `fork` points in the control flow. For example, the state may be duplicated, split, or otherwise updated to note the fork.

The below code promotes `StateT` to be a `Par`-monad transformer:

```
class SplittableState a where
  splitState :: a → (a,a)

instance (SplittableState s, ParMonad m) ⇒
         ParMonad (StateT s m) where
  fork task =
     do s ← State.get
        let (s1,s2) = splitState s
        State.put s2
        lift (fork (do runStateT task s1; return ()))
```

Note that here, `put` and `get` are not LVar operations, but the standard procedures for setting and retrieving the state in a `StateT`.

4.3.3. Determinism guarantee. The `StateT` transformer preserves determinism because it is effectively *syntactic sugar*. That is, `StateT` does not allow one to write any program that could not already be written using the underlying `Par` monad, simply by passing around an extra argument.

This is because `StateT` only provides a *functional* state (an implicit argument and return value), not actual mutable heap locations. Genuine mutable locations in pure computations, on the other hand, require Haskell's ST monad, the safer sister monad to `IO`.

4.3.4. Disjoint parallel update with `ParST`. The LVars model is based on the notion that it is fine for multiple threads to access and update shared memory, so long as updates commute and "build on" one another, only adding information rather than destroying it. Yet it should be possible for threads to update memory destructively, so long as the memory updated by different threads is *disjoint*. This is the approach to deterministic parallelism taken by, for example, Deterministic Parallel Java (DPJ) [**8**], which uses a region-based type and effect system to ensure that each mutable region of the heap is passed

linearly to a thread that then gains exclusive permission to update that region. In order to add this capability to the LVish library, though, we need destructive updates to interoperate with LVar effects. Moreover, we wish to do so at the library level, without requiring language extensions.

Our solution is to provide a `ParST` transformer, a variant of the `StateT` transformer described above. `ParST` allows arbitrarily complex mutable state, such as tuples of vectors (arrays). However, `ParST` enforces the restriction that every memory location in the state is reachable by only one pointer: alias freedom.

Previous approaches to integrating mutable memory with pure functional code (*i.e.*, the ST monad) work with LVish, but only allow thread-private memory. There is no way to operate on the same structure (for instance, on two halves of an array) from different threads. `ParST` exploits the fact that it is perfectly safe to do so as long as the different threads are accessing disjoint parts of the data structure. Below we demonstrate the idea using a simplified convenience module provided alongside the general (`ParST`) library, which handles the specific case of a single vector as the mutable state being shared.

```
runParVecT 10 (
  do -- Fill all 10 slots with "a":
     set "a"
     -- Get a pointer to the state:
     ptr ← reify
     -- Call pre-existing ST code:
     new ← pickLetter ptr
     forkSTSplit (SplitAt 5)
       (write 0 new)
       (write 0 "c")
     -- ptr is again accessible here
     ellipses)
```

This program demonstrates running a parallel, stateful session within a `Par` computation. The shared mutable vector is implicit and global within the monadic `do` block. We fork the control flow of the program with `forkSTSplit`, where `(write 0 new)` and `(write 0 "c")` are the two forked child computations. The `SplitAt` value describes how to partition the state into disjoint pieces: `(SplitAt 5)`

indicates that the element at index 5 in the vector is the "split point", and hence the first child computation passed to `forkSTSplit` may access only the first half of the vector, while the other may access only the second half. (We will see shortly how this generalizes.) Each child computation sees only a *local* view of the vector, so writing `"c"` to index `0` in the second child computation is really writing to index 5 of the global vector.

Ensuring the safety of `ParST` hinges on two requirements:

- *Disjointness:* Any thread can get a direct pointer to its state. In the above example, `ptr` is an `STVector` that can be passed to any standard library procedures in the ST monad. However, it must *not* be possible to access `ptr` from `forkSTSplit`'s child computations. We accomplish this using Haskell's support for higher-rank types,[8] ensuring that accessing `ptr` from a child computation causes a type error. Finally, `forkSTSplit` is a fork-join construct; after it completes the parent thread again has full access to `ptr`.
- *Alias freedom:* Imagine that we expanded the example above to have as its state a *tuple* of two vectors: (v_1, v_2). *If* we allowed the user to supply an arbitrary initial state to their `ParST` computation, then they might provide the state (v_1, v_1), *i.e.*, two copies of the same pointer. This breaks the abstraction, enabling them to reach the same mutable location from multiple threads (by splitting the supposedly-disjoint vectors at a different index). Thus, in LVish, users do not populate the state directly, but only describe a *recipe* for its creation. Each type used as a `ParST` state has an associated type for descriptions of (1) how to create an initial structure, and (2) how to split it into disjoint pieces. We provide a trusted library of instances for commonly used types.

4.3.5. Inter-thread communication. Disjoint state update does not solve the problem of communication between threads. Hence systems built around this idea often include other means for performing reductions, or require "commutativity annotations" for operations such as adding to a set. For instance, DPJ provides a `commuteswith` form for asserting that operations commute with one another to enable

[8] That is, the type of a child computation begins with `(forall s DOT ParST ellipses)`.

concurrent mutation. In LVish, however, such annotations are unnecessary, because LVish already provides a language-level guarantee that all effects commute! Thus, a programmer using LVish with `ParST` can use any LVar to communicate results between threads performing disjoint updates, without requiring trusted code or annotations. Moreover, LVish with `ParST` is unique among deterministic parallel programming models in that it allows both DPJ-style, disjoint destructive parallel updates, and blocking, dataflow-style communication between threads (through LVars).

4.4. The LVish library implementation

LK: In our POPL paper, we discussed the importance of idempotence at the start of the implementation section. However, it seems to me that it should be de-emphasized here. I make a big enough deal of being able to support arbitrary update operations (which aren't idempotent) that it seems disingenuous to make a big thing out of idempotence here. (Also, increment-only counters (which were the whole motivation for generalizing `put` in the first place) aren't atomistic, either! But at least the "Leveraging atoms" stuff doesn't *require* lattices to be atomistic.)

In this section, I describe the internals of the LVish library, including how the library represents LVars and `Par` computations internally and how it implements lattice-generic functions for writing, reading, and freezing LVars, as well as registering handlers with and quiescing LVars. First, though, I discuss a semantic observation about lattices that our implementation makes use of.

4.4.1. Leveraging atoms. Monotonic data structures acquire "pieces of information" over time. In a lattice, the smallest such pieces are called the *atoms* of the lattice: they are elements not equal to $\bot$, but for which the only smaller element is $\bot$. Lattices for which every element is the lub of some set of atoms are called *atomistic*, and in practice most application-specific lattices used by LVish programs have this property—especially those whose elements represent collections.

In general, the LVish primitives allow arbitrarily large queries and updates to an LVar. But for an atomistic lattice, the corresponding data structure usually exposes operations that work at the atom level,

semantically limiting `puts` to atoms, `gets` to threshold sets of atoms, and event sets to sets of atoms. For example, the lattice of finite maps is atomistic, with atoms consisting of all singleton maps (*i.e.*, all key/value pairs). The interface to a finite map usually works at the atom level, allowing addition of a new key/value pair, querying of a single key, or traversals (which we model as handlers) that walk over one key/value pair at a time.

Our implementation is designed to facilitate good performance for atomistic lattices by associating LVars with a set of *deltas* (changes), as well as a lattice. For atomistic lattices, the deltas are essentially just the atoms—for a set lattice, a delta is an element; for a map, a key/value pair. Deltas provide a compact way to represent a change to the lattice, allowing us to easily and efficiently communicate such changes between `puts` and `gets`/handlers.

LK: Someone who is more familiar with the current state of LVish than I am should audit the next few sections and make sure it's still accurate.

4.4.2. Representation choices. LVish uses the following generic representation for LVars:

```
data LVar a d =
     LVar { state :: a,  status :: IORef (Status d) }
```

where the type parameter `a` is the (mutable) data structure representing the lattice, and `d` is the type of deltas for the lattice.[9] The `status` field is a mutable reference that represents the status bit, which says whether or not an LVar is frozen:

```
data Status d = Frozen | Active (B.Bag (Listener d))
```

The status bit of an LVar is tied together with a bag of waiting *listeners*, which include blocked `gets` and handlers; once the LVar is frozen, there can be no further events to listen for.[10] The bag module (imported as `B`) supports atomic insertion and removal, and *concurrent* traversal:

[9] For non-atomistic lattices, we take `a` and `d` to be the same type.
[10] In particular, with one atomic update of the flag we both mark the LVar as frozen and allow the bag to be garbage-collected.

```
put     :: Bag a → a → IO (Token a)
remove  :: Token a → IO ()
foreach :: Bag a → (a → Token a → IO ()) → IO ()
```

Removal of elements is done via abstract *tokens*, which are acquired by insertion or traversal. Updates may occur concurrently with a traversal, but are not guaranteed to be visible to it.

A listener for an LVar is a pair of callbacks, one called when the LVar's lattice value changes, and the other when the LVar is frozen:

```
data Listener d = Listener {
  onUpd :: d → Token (Listener d) → SchedQ → IO (),
  onFrz ::     Token (Listener d) → SchedQ → IO () }
```

The listener is given access to its own token in the listener bag, which it can use to deregister from future events (useful for a `get` whose threshold has been passed). It is also given access to the CPU-local scheduler queue, which it can use to spawn threads.

Internally, the `Par` monad represents computations in continuation-passing style, in terms of their interpretation in the `IO` monad:

```
type ClosedPar = SchedQ → IO ()
type ParCont a = a → ClosedPar
mkPar :: (ParCont a → ClosedPar) → Par lvl a
```

The `ClosedPar` type represents ready-to-run `Par` computations, which are given direct access to the CPU-local scheduler queue. Rather than returning a final result, a completed `ClosedPar` computation must call the scheduler, `sched`, on the queue. A `Par` computation, on the other hand, completes by passing its intended result to its continuation—yielding a `ClosedPar` computation.

4.4.3. Threshold reading. Figure 4.1 gives the implementation for the lattice-generic `getLV` function, which assists data structure authors in writing operations with `get` semantics. In addition to an

```
getLV :: (LVar a d) → (a → Bool → IO (Maybe b))
                    → (d → IO (Maybe b)) → Par lvl b
getLV (LVar{state, status}) gThresh dThresh =
  mkPar $λk q →
    let onUpd d = unblockWhen (dThresh d)
        onFrz   = unblockWhen (gThresh state True)
        unblockWhen thresh tok q = do
          tripped ← thresh
          whenJust tripped $ λb → do
            B.remove tok
            Sched.pushWork q (k b)
    in do
      curStat ← readIORef status
      case curStat of
        Frozen → do -- no further deltas can arrive!
          tripped ← gThresh state True
          case tripped of
            Just b  → exec (k b) q
            Nothing → sched q
        Active ls → do
          tok ← B.put ls (Listener onUpd onFrz)
          frz ← isFrozen status -- must recheck after enrolling listener
          tripped ← gThresh state frz
          case tripped of
            Just b  → do
              B.remove tok -- remove the listener
              k b q        -- execute our continuation
            Nothing → sched q
```

Figure 4.1. Implementation of the `getLV` function.

LVar, it takes two *threshold functions* as arguments, one for global state and one for deltas.[11] The *global threshold* `gThresh` is used to initially check whether the LVar is above some lattice value(s) by global inspection; the extra boolean argument gives the frozen status of the LVar. The *delta threshold* `dThresh`

[11] In this sense, the way that threshold reads are implemented in LVish of is in fact quite close to the semantic notion of *threshold functions* that I described in Section 2.6.3.

checks whether a particular update takes the state of the LVar above some lattice state(s). Both functions return `Just r` if the threshold has been passed, where `r` is the result of the read. To continue our running example of finite maps with key/value pair deltas, we can use `getLV` internally to build the following `getKey` function that is exposed to application writers:

```
-- Wait for the map to contain a key; return its value
getKey key mapLV = getLV mapLV gThresh dThresh where
  gThresh m frozen = lookup key m
  dThresh (k,v) | k == key  = return (Just v)
                | otherwise = return Nothing
```

where `lookup` imperatively looks up a key in the underlying map.

The challenge in implementing `getLV` is the possibility that a *concurrent* `put` will push the LVar over the threshold. To cope with such races, `getLV` employs a somewhat pessimistic strategy: before doing anything else, it enrolls a listener on the LVar that will be triggered on any subsequent updates. If an update passes the delta threshold, the listener is removed, and the continuation of the `get` is invoked, with the result, in a new lightweight thread. *After* enrolling the listener, `getLV` checks the *global* threshold, in case the LVar is already above the threshold. If it is, the listener is removed, and the continuation is launched immediately; otherwise, `getLV` invokes the scheduler, effectively treating its continuation as a blocked thread.

By doing the global check only after enrolling a listener, `getLV` is sure not to miss any threshold-passing updates. It does *not* need to synchronize between the delta and global thresholds: if the threshold is passed just as `getLV` runs, it might launch the continuation twice (once via the global check, once via delta), but by idempotence this does no harm. This is a performance tradeoff: we avoid imposing extra synchronization on *all* uses of `getLV` at the cost of some duplicated work in a rare case. We can easily provide a second version of `getLV` that makes the alternative tradeoff, but as we will see below, idempotence plays an *essential* role in the analogous situation for handlers.

```
putLV :: LVar a d → (a → IO (Maybe d)) → Par lvl ()
putLV (LVar{state, status}) doPut = mkPar $ λk q → do
  Sched.mark q  -- publish our intent to modify the LVar
  delta   ← doPut state      -- possibly modify LVar
  curStat ← readIORef status -- read while q is marked
  Sched.clearMark q          -- retract our intent
  whenJust delta $ λd → do
    case curStat of
      Frozen → error "Attempt to change a frozen LVar"
      Active listeners → B.foreach listeners $
        λ(Listener onUpd _) tok → onUpd d tok q
  k () q
```

Figure 4.2. Implementation of the putLV function.

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

Figure 4.3. Implementation of the freezeLV function.

4.4.4. Putting and freezing. Figures 4.2 and 4.3 respectively give the implementations for the lattice-generic putLV and freezeLV functions. The putLV function is used to build operations with put semantics. It takes an LVar and an *update function* doPut that performs the put on the underlying data structure, returning a delta if the put actually changed the data structure. If there is such a delta, putLV subsequently invokes all currently-enrolled listeners on it.

The implementation of `putLV` is complicated by another race, this time with freezing. If the `put` is nontrivial (*i.e.*, if it changes the value of the LVar), the race can be resolved in two ways. Either the freeze takes effect first, in which case the `put` must fault, or else the `put` takes effect first, in which case both succeed. Unfortunately, we have no means to both check the frozen status *and* attempt an update in a single atomic step.[12]

Our basic approach is to ask forgiveness, rather than permission: we eagerly perform the `put`, and only afterwards check whether the LVar is frozen. Intuitively, this is allowed because *if* the LVar is frozen, the `Par` computation is going to terminate with an exception—so the effect of the `put` cannot be observed!

Unfortunately, it is not enough to *just* check the status bit for frozenness afterward, for a rather subtle reason: suppose the `put` is executing concurrently with a `get` which it causes to unblock, and that the getting thread subsequently freezes the LVar. In this case, we *must* treat the `freeze` as if it happened after the `put`, because the `freeze` could not have occurred had it not been for the `put`. But, by the time `putLV` reads the status bit, it may already be set, which naively would cause `putLV` to fault.

To guarantee that such confusion cannot occur, we add a *marked* bit to each CPU scheduler state. The bit is set (using `Sched.mark`) prior to a `put` being performed, and cleared (using `Sched.clear`) only *after* `putLV` has subsequently checked the frozen status. On the other hand, `freezeLV` waits until it has observed a (transient!) clear mark bit on every CPU (using `Sched.awaitClear`) before actually freezing the LVar. This guarantees that any `puts` that *caused* the freeze to take place check the frozen status *before* the freeze takes place; additional `puts` that arrive concurrently may, of course, set a mark bit again after `freezeLV` has observed a clear status.

The proposed approach requires no barriers or synchronization instructions (assuming that the `put` on the underlying data structure acts as a memory barrier). Since the mark bits are per-CPU flags, they can generally be held in a core-local cache line in exclusive mode—meaning that marking and clearing them

[12] While we could require the underlying data structure to support such transactions, doing so would preclude the use of existing lock-free data structures, which tend to use a single-word compare-and-set operation to perform atomic updates. Lock-free data structures routinely outperform transaction-based data structures [23].

is extremely cheap. The only time that the busy flags can create cross-core communication is during `freezeLV`, which should only occur once per LVar computation.

One final point: unlike `getLV` and `putLV`, which are polymorphic in their determinism level, `freezeLV` is statically `QuasiDet`, because a computation that performs freezes is necessarily quasi-deterministic.

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

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

where `Counter` is a simple counter supporting atomic increment, decrement, and checks for equality with zero.[13] We use the counter to track the number of currently-executing callbacks, which we can use to implement `quiesce`. A handler pool also keeps a bag of threads that are blocked waiting for the pool to reach a quiescent state.

We create a pool using `newPool` (of type `Par lvl HandlerPool`), and implement quiescence testing as follows:

```
quiesce :: HandlerPool → Par lvl ()
quiesce hp@(HandlerPool cnt bag) = mkPar $ λk q → do
  tok ← B.put bag (k ())
  quiescent ← poll cnt
  if quiescent then do B.remove tok; k () q
               else sched q
```

where the `poll` function indicates whether `cnt` is (transiently) zero. Note that we are following the same listener-enrollment strategy as in `getLV`, but with `blocked` acting as the bag of listeners.

Finally, `addHandler` has the following interface:

```
addHandler ::
     Maybe HandlerPool              -- Pool to enroll in
```

[13] One can use a high-performance *scalable non-zero indicator* [19] to implement `Counter`, but we have not yet done so.

```
  → LVar a d                          -- LVar to listen to
  → (a → IO (Maybe (Par lvl ()))) -- Global callback
  → (d → IO (Maybe (Par lvl ()))) -- Delta callback
  → Par lvl ()
```

As with `getLV`, handlers are specified using both global and delta threshold functions. Rather than returning results, however, these threshold functions return computations to run in a fresh lightweight thread if the threshold has been passed. Each time a callback is launched, the callback count is incremented; when it is finished, the count is decremented, and if zero, all threads blocked on its quiescence are resumed.

4.4.6. Discussion: leveraging idempotency. While I have emphasized the commutativity of least upper bounds, they also provide another important property: *idempotence*, meaning that $d \sqcup d = d$ for any element d. In LVar terms, repeated (least-upper-bound) `puts` or `freezes` have no effect, and so, in an LVish program that restricts itself to least-upper-bound `puts` (instead of arbitrary update operations, which are not necessarily idempotent), the result is that $e; e$ behaves the same as e for any LVish expression e.

Idempotence has already been recognized as a useful property for work-stealing scheduling [37]: if the scheduler is allowed to occasionally duplicate work, it is possible to substantially save on synchronization costs. For LVish computations that are guaranteed to be idempotent, we could use such a scheduler (although the existing implementation uses the standard Chase-Lev deque [15]).

Moreover, idempotence also helps us deal with races between writes and calls to `addHandler`. The implementation of `addHandler` is very similar to `getLV`, but there is one important difference: handler callbacks must be invoked for *all* events of interest, not just a single threshold. Thus, the `Par` computation returned by the global threshold function should execute its callback on, *e.g.*, all available atoms. Likewise, we do not remove a handler from the bag of listeners when a single delta threshold is passed; handlers listen continuously to an LVar until it is frozen. We might, for example, expose the following `foreach` function for a finite map:

```
foreach mh mapLV cb = addHandler mh lv gThresh dThresh
  where
    dThresh (k,v) = return (Just (cb k v))
    gThresh mp    = traverse mp (λ(k,v) → cb k v) mp
```

Here, idempotence really pays off: without it, we must synchronize to ensure that no callbacks are duplicated between the global threshold (which may or may not see concurrent additions to the map) and the delta threshold (which will catch all concurrent additions).

Naturally, it is best to pay the aforementioned synchronization overhead only when required. This requires static information about whether a given program uses non-idempotent writes. Fortunately, LVish's fine-grained effect-tracking capability can provide precisely this information. We refer to a write that is specifically non-idempotent as a *bump*, and the `HasBump` effect level constraint says that a `Par` computation is allowed to perform such writes. For example, an increment-only counter might have an `incrCounter` operation with the following signature:

```
incrCounter :: HasBump e ⇒ Counter s → Par s e ()
```

LK: This is from the PLDI paper, but shouldn't it also have `NoPut` as a constraint? Also, do we assume`HasPut` and `NoBump` by default?

LK: There should really be more in this section about how the information provided by `HasBump` or `NoBump`, etc., is actually communicated to and used by the runtime system. But I don't actually know how that works! :(Ryan and Aaron, I need your help here.

4.5. Case study: parallelizing k-CFA with LVish

LVish is designed to be particularly applicable to (1) parallelizing complicated algorithms on structured data that pose challenges for other deterministic paradigms, and (2) composing pipeline-parallel stages of computation (each of which may be internally parallelized). In this section, I describe a case study

that fits this mold: *parallelized control-flow analysis*. I discuss the process of porting a sequential implementation of a k-CFA static program analysis to a parallel implementation using LVish.

The *k-CFA* analyses provide a hierarchy of increasingly precise methods to compute the flow of values to expressions in a higher-order language. For this case study, we began with a sequential implementation of k-CFA translated to Haskell from a version by Might [38].[14] The algorithm processes expressions written in a continuation-passing-style λ-calculus. It resembles a nondeterministic abstract interpreter in which stores map addresses to *sets* of abstract values, and function application entails a cartesian product between the operator and operand sets.

Further, an address models not just a static variable, but includes a fixed k-size window of the calling history to get to that point (the k in k-CFA).

Taken together, the current redex, environment, store, and call history make up the abstract state of the program, and the goal is to explore a graph of these abstract states in order to discover the flow of control of a program without actually running it. This graph-exploration phase is followed by a second, summarization phase that combines all the information discovered into one store.

4.5.1. k-CFA phase 1: breadth-first exploration. The following function from the original, sequential version of the analysis expresses the heart of the search process:

```
explore :: Set State → [State] → Set State
explore seen [] = seen
explore seen (todo:todos)
  | todo ∈ seen = explore seen todos
  | otherwise   = explore (insert todo seen)
                          (toList (next todo) ++ todos)
```

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

This code uses idiomatic Haskell data types like `Data.Set` and lists. However, it presents a dilemma with respect to exposing parallelism. Consider attempting to parallelize `explore` using purely functional parallelism with futures—for instance, using the Haskell Strategies library [34]. An attempt to compute the next states in parallel would seem to be thwarted by the main thread rapidly forcing each new state to perform the seen-before check, `todo` $\in$`seen`. There is no way for independent threads to "keep going" further into the graph; rather, they check in with `seen` after one step.

We confirmed this prediction by adding a parallelism annotation from the aforementioned Strategies library:

```
withStrategy (parBuffer 8 rseq) (next todo)
```

The GHC runtime reported that 100% of created futures were "duds"—that is, the main thread forced them before any helper thread could assist. Changing `rseq` to `rdeepseq` exposed a small amount of parallelism—238/5000 futures were successfully executed in parallel—yielding no actual speedup.

4.5.2. k-CFA phase 2: summarization. The first phase of the algorithm produces a large set of states, with stores that need to be joined together in the summarization phase. When one phase of a computation produces a large data structure that is immediately processed by the next phase, lazy languages can often achieve a form of pipelining "for free". This outcome is most obvious with *lists*, where the head element can be consumed before the tail is computed, offering cache-locality benefits. Unfortunately, when processing a pure `Set` or `Map` in Haskell, such pipelining is not possible, since the data structure is internally represented by a balanced tree whose structure is not known until all elements are present. Thus phase 1 and phase 2 cannot overlap in the purely functional version—but they will in the LVish version, as we will see. In fact, in LVish we will be able to achieve partial deforestation in addition to pipelining. Full deforestation in this application is impossible, because the `Sets` in the implementation serve a memoization purpose: they prevent repeated computations as we traverse the graph of states.

4.5.3. Porting to LVish. Our first step in parallelizing the original k-CFA implementation was a *verbatim* port to LVish: that is, we changed the original, purely functional program to allocate a new LVar for each new set or map value in the original code. This was done simply by changing two types, `Set` and `Map`, to their LVar counterparts, `ISet` and `IMap`. In particular, a store maps a program location (with context) onto a set of abstract values:

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

Next, we replaced allocations of containers, and `map`/`fold` operations over them, with the analogous operations on their LVar counterparts. The `explore` function above was replaced by the simple graph traversal function from Section 3.1.4. These changes to the program were mechanical, including converting pure to monadic code. Indeed, the key insight in doing the verbatim port to LVish was to consume LVars as if they were pure values, ignoring the fact that an LVar's contents are spread out over space and time and are modified through effects.

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

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

While this code can be read in terms of traditional parallel nested loops, it in fact creates a network of handlers that convey incremental updates from one LVar to another, in the style of data-flow networks.

That means, in particular, that computations in a pipeline can *immediately* begin reading results from containers (*e.g.*, `storeFin`), long before their contents are final.

The LVish version of k-CFA contains eleven occurrences of `forEach`, as well as a few *cartesian-product* operations. The cartesian products serve to apply functions to combinations of all possible values that arguments may take on, greatly increasing the number of handler events in circulation. Moreover, chains of handlers registered with `forEach` result in cascades of events through six or more handlers. The runtime behavior of these would be difficult to reason about. Fortunately, the programmer can largely ignore the temporal behavior of their program, since all LVish effects commute—rather like the way in which a lazy functional programmer typically need not think about the order in which thunks are forced at runtime.

Finally, there is an optimization benefit to using handlers. Normally, to flatten a nested data structure such as `[[[Int]]]` in a functional language, one would need to flatten one layer at a time and allocate a series of temporary structures. The LVish version avoids this; for example, in the code for `summarize` above, three `forEach` invocations are used to traverse a triply-nested structure, and yet the side effect in the innermost handler directly updates the final accumulator, `storeFin`.

4.5.4. Flipping the switch: the advantage of sharing. The verbatim port uses LVars poorly: copying them repeatedly and discarding them without modification. This effect overwhelms the benefits of partial deforestation and pipelining, and the verbatim LVish port has a small performance overhead relative to the original. But not for long!

The most clearly unnecessary operation in the verbatim port is in the `next` function. Like the purely functional program from which it was ported, it creates a fresh store to extend with new bindings as we take each step through the state space graph:

```
store' ← IM.copy store
```

Of course, a "copy" for an LVar is persistent: it is just a handler that forces the copy to receive everything the original does. But in LVish, it is also trivial to *entangle* the parallel branches of the search, allowing them to share information about bindings, simply by *not* creating a copy:

```
let store' = store
```

This one-line change speeds up execution by up to $25\times$ *on one core*. The lesson here is that, although pure functional parallel programs are guaranteed to be deterministic, the overhead of allocation and copying in an idiomatic pure functional program can overwhelm the advantages of parallelism. In the LVish version, the ability to use shared mutable data structures—even though they are only mutable in the extremely restricted and determinism-preserving way that LVish allows—affords a significant speedup even when the code runs sequentially. The effect is then multiplied as we add parallel resources: the asynchronous, `ISet`-driven parallelism enables parallel speedup for a total of up to $202\times$ total improvement over the purely functional version.

4.5.5. Parallel speedup results. We implemented two versions of the k-CFA algorithm using set data structures that the LVish library provides. The first of these, `PureSet`, exported by the `Data.LVar.PureSet` module) uses the LVish library's reference implementation of a set, which uses a pure `Data.Set` wrapped in a mutable container. The other, `SLSet`, exported by `Data.LVar.SLSet`, is a lock-free set based on concurrent skip lists [28].[15]

We evaluated both the `PureSet`-based and `SLSet`-based k-CFA implementations on two benchmarks. For the first, we used a version of the "blur" benchmark from a recent paper on k-CFA by Earl *et al*. [18]. In general, it proved difficult to generate example inputs to k-CFA that took long enough to be candidates for parallel speedup; we were, however, able to "scale up" the blur benchmark by replicating the code N times, feeding one into the continuation argument for the next. For our second benchmark, we

[15] LVish also provides analogous reference and lock-free implementations of maps (`PureMap` and `SLMap`). In fact, LVish is the first project to incorporate *any* lock-free data structures in Haskell, which required solving some unique problems pertaining to Haskell's laziness and the GHC compiler's assumptions regarding referential transparency.LK: Question for Ryan: "first project to incorporate any lock-free data structures in Haskell" seems like a strong statement that I don't want to let go unsupported—is there anything I can point to about this? Even just a blog post?

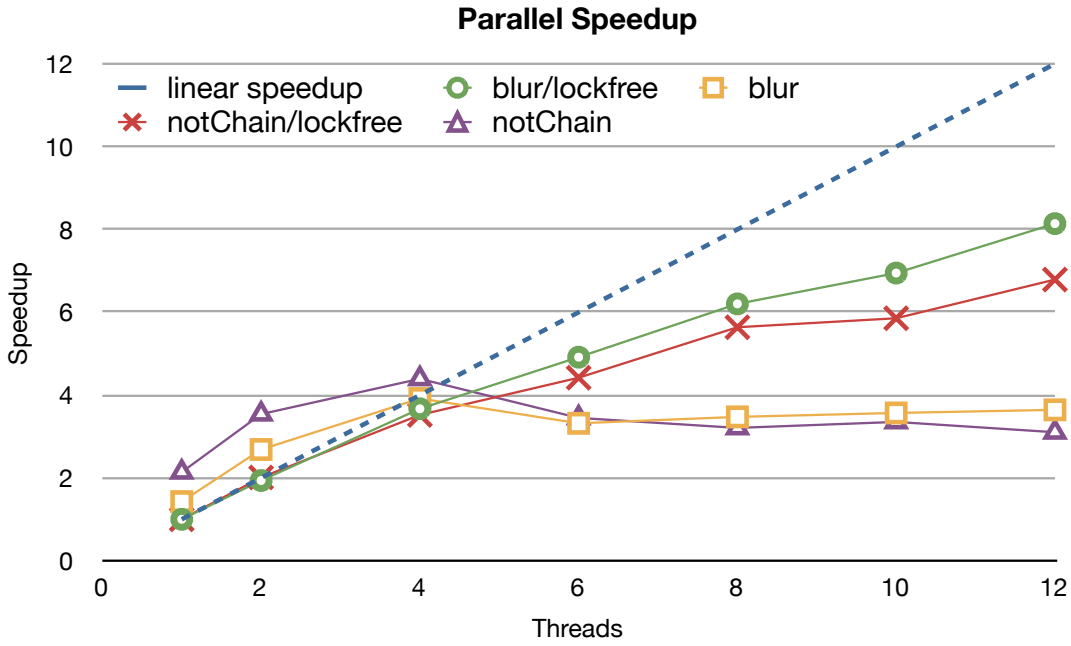

Figure 4.4. Parallel speedup for the "blur" and "notChain" benchmarks. Speedup is normalized to the sequential times for the *lock-free* versions (5.21s and 9.83s, respectively). The normalized speedups are remarkably consistent for the lock-free version between the two benchmarks. But the relationship to the original, purely functional version is quite different: at 12 cores, the lock-free LVish version of "blur" is $202\times$ faster than the original, while "notChain" is only $1.6\times$ faster, not gaining anything from sharing rather than copying stores due to a lack of fan-out in the state graph.

ran the k-CFA analysis on a program that was simply a long chain of 300 "not" functions (using a CPS conversion of the Church encoding for booleans). This latter benchmark, which we call "notChain", has a small state space of large states with many variables (600 states and 1211 variables), and was specifically designed to negate the benefits of our sharing approach.

Figure 4.4 shows the parallel speedup results of our experiments on a twelve-core machine.[16] (We used $k = 2$ for the benchmarks in this section.) The lines labeled "blur" and "blur/lockfree" show the parallel

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

speedup of the "blur" benchmark for the `PureSet`-based implementation and `SLSet`-based implementation of k-CFA, respectively, and the lines labeled "notChain" and "notChain/lockfree" show parallel speedup of the "notChain" benchmark for the `PureSet`-based and `SLSet`-based implementations, respectively.

The results for the `PureSet`-based implementations are normalized to the same baseline as the results for the `SLSet`-based implementations at one core. At one and two cores, the `SLSet`-based k-CFA implementation (shown in green) is 38% to 43% slower than the `PureSet`-based implementation (in yellow) on the "blur" benchmark. The `PureSet`-based implementation, however, stops scaling after four cores. Even at four cores, variance is high in the `PureSet`-based implementation (min/max 0.96s / 1.71s over 7 runs). Meanwhile, the `SLSet`-based implementation continues scaling and achieves an $8.14\times$ speedup on twelve cores (0.64s at 67% GC productivity).

Of course, it is unsurprising that using an efficient, concurrent, lock-free shared data structure results in a better parallel speedup; rather, the interesting thing about these results is that despite its determinism guarantee, *there is nothing about the LVars model that precludes using such data structures*. Any data structure that has the semantics of an LVar is fine. Part of the benefit of LVish is to allow parallel programs to make use of lock-free data structures while retaining the determinism guarantee of LVars, in much the same way that the ST monad allows access to efficient in-place array computations.

4.6. Case study: parallelizing PhyBin with LVish

One reason why we might want guaranteed-deterministic software is for the sake of scientific repeatability: in bioinformatics, for example, we would expect an experiment on the same data set to produce the same result on every run. In this section, I describe our experience using the LVish library to parallelize *PhyBin*, a bioinformatics application for comparing phylogenetic trees. A *phylogenetic tree* represents a possible ancestry for a set of N species. Leaf nodes in the tree are labeled with species' names, and

the structure of the tree represents a hypothesis about common ancestors. For a variety of reasons, biologists often end up with many alternative trees, whose relationships they need to then analyze.

PhyBin [39] is a medium-sized (3500-line) bioinformatics program implemented in Haskell[17] for this purpose, initially released in 2010. The primary output of the software is a hierarchical clustering of the input tree set (that is, a tree of trees), but most of its computational effort is spent computing an $N \times N$ distance matrix that records the *edit distance* between each pair of input trees. It is this distance computation that we parallelize in our case study.

4.6.1. Computing all-to-all tree edit distance. The distance metric itself is called *Robinson-Foulds* (RF) distance, and the fastest algorithm for all-to-all RF distance computation is Sul and Williams' *HashRF* algorithm [48], introduced by a software package of the same name.[18] The HashRF software package is written in C++ and is about 2-3$\times$ as fast as PhyBin. Both packages are dozens or hundreds of times faster than the more widely-used software that computes RF distance matrices, such as PHYLIP[19] [21] and DendroPy[20] [47]. These slower packages use $\frac{N^2-N}{2}$ full applications of the distance metric, which has poor locality in that it reads all trees in from memory $\frac{N^2-N}{2}$ times.

To see how the HashRF algorithm improves on this, consider that each edge in an unrooted phylogenetic tree can be seen as partitioning the tree's nodes into two disjoint sets, according to the two subtrees that those nodes would belong to if the edge were deleted. For example, if a tree has nodes $\{a, b, c, d, e\}$, one bipartition or "split" might be $\{\{a, b\}, \{c, d, e\}\}$, while another might be $\{\{a, b, c\}, \{d, e\}\}$. A tree can therefore be encoded as a *set of bipartitions* of its nodes. Furthermore, once trees are encoded as sets of bipartitions, we can compute the edit distance between trees (that is, the number of operations required to transform one tree into the other) by computing the *symmetric set difference* between sets of bipartitions, and we can do so using standard set data structures.

[17] Available at http://hackage.haskell.org/package/phybin.
[18] Available at https://code.google.com/p/hashrf/.
[19] Available at http://evolution.genetics.washington.edu/phylip.html.
[20] Available at http://pythonhosted.org/DendroPy/.

```
(1) for t ∈ alltrees:
      for bip ∈ t:
        insert(splitsmap, (t, bip))
(2) for (_, tree_set) ∈ splitsmap:
      for t1 ∈ alltrees:
        for t2 ∈ alltrees:
          if t1 ∈ tree_set `xor` t2 ∈ tree_set
          then increment(distancematrix[t1,t2])
```

Figure 4.5. Pseudocode of the HashRF algorithm for computing a tree edit distance matrix. `alltrees` is the set of trees, represented as sets of bipartitions; `splitsmap` and `distancematrix` are global variables, defined elsewhere. `splitsmap` maps bipartitions to sets of trees in which they occur. The comparison uses ‘xor‘ because the RF distance between two trees is defined as the number of bipartitions implied by exactly one of the two trees being compared.

The HashRF algorithm makes use of this fact and adds a clever trick that greatly improves locality. Before computing the actual distances between trees, it populates a table, the “splits map”, which maps each observed bipartition to a set of IDs of trees that contain that bipartition. The second phase of the algorithm, which actually computes the $N \times N$ distance matrix, does so by iterating through each entry in the splits map. For each such entry, for each pair of tree IDs, it checks whether exactly one of those tree IDs is in the splits map entry, and if so, increments the appropriate distance matrix entry by one.

Figure 4.5 gives psuedocode for the HashRF algorithm. The second phase of the algorithm is still $O(N^2)$, but it only needs to read from the much smaller `tree_set` during this phase. All loops in Figure 4.5 are potentially parallel.

LK: Question for Ryan: shouldn’t it be “insert(splitsmap, (bip, t))” since it maps bipartitions to trees, not the other way around? Or, really, it maps bipartitions to sets of tree IDs, so maybe this could be made more obvious in the pseudocode somehow.

4.6.2. Parallelizing the HashRF algorithm with LVish. In the original PhyBin source code, the type of the splits map is:

```
type BipTable = Map DenseLabelSet (Set TreeID)
```

Here, a `DenseLabelSet` encodes an individual bipartition as a bit vector. PhyBin uses purely functional data structures for the `Map` and `Set` types, whereas the HashRF implementation uses a mutable hash table. Yet in both cases, these structures grow monotonically during execution, making PhyBin a good candidate for parallelization with LVish. The splits map created in the first phase is a map of sets, which are directly replaced by their LVar counterparts, and the distance matrix created in the second phase is a vector of monotonically increasing counters. Furthermore, the second phase of the algorithm can be pipelined with the first, because as soon as one of the rows of the splits map is complete, the second phase can start working on that row. LK: Question for Ryan: is that last bit actually true? if so, why is this pipelining possible – don't we have to freeze the splits map before we can read from it?

In fact, the parallel port of PhyBin using LVish was so straightforward that, after reading the code, parallelizing the first phase of the algorithm took only 29 minutes.[21] Once the second phase was ported, the distance computation sped up by a factor of $3.35\times$ on 8 cores. Table 4.1 shows the results of a running time comparison of the parallelized PhyBin with DendroPy, PHYLIP, and HashRF. We first benchmarked PhyBin against DendroPy and PHYLIP using a set of 100 trees with 150 leaves each. The table shows the time it took in each case to fill in the all-to-all tree edit distance matrix and get an answer back. PhyBin was much faster than these two alternatives.

Then, to compare PhyBin with HashRF, we used a set of 1000 trees with 150 leaves each. HashRF took about 1.7 seconds to process these 1000 trees, but since is a single-threaded program, so adding cores does not offer any speedup. PhyBin, while slower than HashRF on one core, taking about 4.7 seconds, speeds up as we add cores and eventually overtakes HashRF, running in about 1.4 seconds on 8 cores. This is exactly the sort of situation in which we would like to use LVish—to achieve modest speedups for

[21] Git commit range: https://github.com/rrnewton/PhyBin/compare/5cbf7d26c07a...6a05cfab490a7a.

Trees	Species	PhyBin			DendroPy	PHYLIP
100	150	0.269			22.1	12.8
		PhyBin 1, 2, 4, 8 core				HashRF
1000	150	4.7	3	1.9	1.4	1.7

Table 4.1. PhyBin performance comparison with DendroPy, PHYLIP, and HashRF. All times in seconds.

modest effort, in programs with complicated data structures (and high allocation rates), and without changing the determinism guarantee of the original functional code.

CHAPTER 5

Deterministic threshold queries of distributed data structures

Distributed systems typically involve *replication* of data objects across a number of physical locations. Replication is of fundamental importance in such systems: it makes them more robust to data loss and allows for good data locality. But the well-known *CAP theorem* [25, 10] of distributed computing imposes a trade-off between *consistency*, in which every replica sees the same data, and *availability*, in which all data is available for both reading and writing by all replicas. *Highly available* distributed systems, such as Amazon's Dynamo key-value store [17], relax strong consistency in favor of *eventual consistency* [52], in which replicas need not agree at all times. Instead, updates execute at a particular replica and are sent to other replicas later. All updates eventually reach all replicas, albeit possibly in different orders. Informally speaking, eventual consistency says that if updates stop arriving, all replicas will *eventually* come to agree.

Although giving up on strong consistency makes it possible for a distributed system to offer high availability, even an eventually consistent system must have some way of resolving conflicts between replicas that differ. One approach is to try to determine which replica was written most recently, then declare that replica the winner. But, even in the presence of a way to reliably synchronize clocks between replicas and hence reliably determine which replica was written most recently, having the last write win might not make sense from a *semantic* point of view. For instance, if a replicated object represents a set, then, depending on the application, the appropriate way to resolve a conflict between two replicas could be to take the set union of the replicas' contents. Such a conflict resolution policy might be more appropriate than a "last write wins" policy for, say, a object representing the contents of customer shopping carts for an online store [17].

Implementing application-specific conflict resolution policies in an ad-hoc way for every application is tedious and error-prone.[1] Fortunately, we need not implement them in an ad-hoc way. Shapiro *et al.*'s *convergent replicated data types* (CvRDTs) [**45**, **44**] provide a simple mathematical framework for reasoning about and enforcing the eventual consistency of replicated objects, based on viewing replica states as elements of a lattice and replica conflict resolution as the lattice's join operation.

Like LVars, CvRDTs are data structures whose states are elements of an application-specific lattice, and whose contents can only grow with respect to the given lattice. Although LVars and CvRDTs were developed independently, both models leverage the mathematical properties of join-semilattices to ensure that a property of the model holds—determinism in the case of LVars; eventual consistency in the case of CvRDTs.

CvRDTs offer a simple and theoretically-sound approach to eventual consistency. However, with CvRDTs (and unlike with LVars), it is still possible to observe inconsistent *intermediate* states of replicated shared objects, and high availability requires that reads return a value immediately, even if that value is stale.

In practice, applications call for both strong consistency and high availability at different times [**50**], and increasingly, they support consistency choices at the granularity of individual queries, not that of the entire system. For example, the Amazon SimpleDB database service gives customers the choice between eventually consistent and strongly consistent read operations on a per-read basis [**53**].

Ordinarily, strong consistency is a global property: all replicas agree on the data. When we make consistency choices at a *per-query* granularity, though, a global strong consistency property need not hold. I define a *strongly consistent query* to be one that, if it returns a result x when executed at a replica i:

- will always return x on subsequent executions at i, and
- will *eventually* return x when executed at *any* replica, and will *block* until it does so.

[1] Indeed, as the developers of Dynamo have noted [**17**], Amazon's shopping cart presents an anomaly whereby removed items may re-appear in the cart!

That is, a strongly consistent query of a distributed data structure, if it returns, will return a result that is a *deterministic* function of all updates to the data structure in the entire distributed execution, regardless of when the query executes or which replica it occurs on.

Traditional CvRDTs only support eventually consistent queries. We could get strong consistency by waiting until all replicas agree before allowing a query to return—but in practice, such agreement may never happen. In this chapter, I present an alternative approach to supporting strongly consistent queries of CvRDTs that takes advantage of their existing lattice structure and does *not* require waiting for all replicas to agree. To do so, I take inspiration from LVar-style threshold reads. I show how to extend CvRDTs to support deterministic, strongly consistent queries, which I call *threshold queries*. After reviewing the fundamentals of CvRDTs in Section 5.1, I introduce CvRDTs extended with threshold queries (Section 5.2) and prove that threshold queries in our extended model are strongly consistent queries (Section 5.3). That is, I show that a threshold query that returns an answer when executed on a replica will return the same answer every subsequent time that it is executed on that replica, and that executing that threshold query on a different replica will eventually return the same answer, and will block until it does so. It is therefore impossible to observe different results from the same threshold query, whether at different times on the same replica or whether on different replicas.

5.1. Background: CvRDTs and eventual consistency

Shapiro *et al*. [**45**, **44**] define an *eventually consistent* object as one that meets three conditions. One of these conditions is the property of *convergence*: all correct replicas of an object at which the same updates have been delivered eventually have equivalent state. The other two conditions are *eventual delivery*, meaning that all replicas receive all update messages, and *termination*, meaning that all method executions terminate (we discuss methods in more detail below).

Shapiro *et al*. further define a *strongly eventually consistent* (SEC) object as one that is eventually consistent and, in addition to being merely convergent, is *strongly convergent*, meaning that correct replicas

at which the same updates have been delivered have equivalent state.[2] A *conflict-free replicated data type* (CRDT), then, is a data type (*i.e.*, a specification for an object) satisfying certain conditions that are sufficient to guarantee that the object is SEC. (The term "CRDT" is used interchangeably to mean a specification for an object, or an object meeting that specification.)

There are two "styles" of specifying a CRDT: *state-based*, also known as *convergent*[3]; or *operation-based* (or "op-based"), also known as *commutative*. CRDTs specified in the state-based style are called *convergent replicated data types*, abbreviated *CvRDTs*, while those specified in the op-based style are called *commutative replicated data types*, abbreviated *CmRDTs*. Of the two styles, we focus on the CvRDT style in this paper because CvRDTs are lattice-based data structures and therefore amenable to threshold queries—although, as Shapiro *et al*. show, CmRDTs can emulate CvRDTs and vice versa.

5.1.1. State-based objects. The Shapiro *et al*. model specifies a *state-based object* as a tuple (S, s^0, q, u, m), where S is a set of states, s^0 is the initial state, q is the *query method*, u is the *update method*, and m is the *merge method*. Objects are replicated across some finite number of processes, with one replica at each process, and each replica begins in the initial state s^0. The state of a local replica may be queried via the method q and updated via the method u. Methods execute locally, at a single replica, but the merge method m can merge the state from a remote replica with the local replica. The model assumes that each replica sends its state to the other replicas infinitely often, and that eventually every update reaches every replica, whether directly or indirectly.

The assumption that replicas send their state to one another "infinitely often" refers not to the *frequency* of these state transmissions; rather; it says that, regardless of what event (such as an update, via the u method) occurs at a replica, a state transmission is guaranteed to occur after that event. We

[2] Strong eventual consistency is not to be confused with strong consistency: it is the combination of eventual consistency and strong convergence. Contrast with ordinary convergence, in which replicas only *eventually* have equivalent state. In a strongly convergent object, knowing that the same updates have been delivered to all correct replicas is sufficient to ensure that those replicas have equivalent state, whereas in an object that is merely convergent, there might be some further delay before all replicas agree.

[3] There is a potentially misleading terminology overlap here: the definitions of convergence and strong convergence above pertain not only to CvRDTs (where the C stands for "Convergent"), but to *all* CRDTs.

can therefore conclude that all updates eventually reach all replicas in a state-based object, meeting the "eventual delivery" condition discussed above. However, we still have no guarantee of strong convergence or even convergence. This is where Shapiro *et al.*'s notion of a CvRDT comes in: a state-based object that meets the criteria for a CvRDT is guaranteed to have the strong-convergence property.

A *state-based* or *convergent* replicated data type (CvRDT) is a state-based object equipped with a partial order $\leq$, written as a tuple $(S, \leq, s^0, q, u, m)$, that has the following properties:

- S forms a join-semilattice ordered by $\leq$.
- The merge method m computes the join of two states with respect to $\leq$.
- State is *inflationary* across updates: if u updates a state s to s', then $s \leq s'$.

Shapiro *et al*. show that a state-based object that meets the criteria for a CvRDT is strongly convergent. Therefore, given the eventual delivery guarantee that all state-based objects have, and given an additional assumption that all method executions terminate, a state-based object that meets the criteria for a CvRDT is SEC [45].

5.1.2. Discussion: the need for inflationary updates. Although CvRDT updates are required to be inflationary, we note that it is not clear that inflationary updates are necessarily required for convergence. Consider, for example, a scenario in which replicas 1 and 2 both have the state $\{a, b\}$. Replica 1 updates its state to $\{a\}$, a non-inflationary update, and then sends its updated state to replica 2. Replica 2 merges the received state $\{a\}$ with $\{a, b\}$, and its state remains $\{a, b\}$. Then replica 2 sends its state back to replica 1; replica 1 merges $\{a, b\}$ with $\{a\}$, and its state becomes $\{a, b\}$. The non-inflationary update has been lost, and was, perhaps, nonsensical—but the replicas are nevertheless convergent.

However, once we introduce threshold queries of CvRDTs, as we will do in the following section, inflationary updates become *necessary* for the determinism of threshold queries. This is because a non-inflationary update could cause a threshold query that had been unblocked to block again, and so arbitrary interleaving of non-inflationary writes and threshold queries would lead to nondeterministic

behavior. Therefore the requirement that updates be inflationary will not only be sensible, but actually crucial.

5.2. Adding threshold queries to CvRDTs

In Shapiro *et al.*'s CvRDT model, the query operation q reads the exact contents of its local replica, and therefore different replicas may see different states at the same time, if not all updates have been propagated yet. That is, it is possible to observe intermediate states of a CvRDT replica. Such intermediate observations are not possible with threshold queries. In this section, we show how to extend the CvRDT model to accommodate threshold queries.

5.2.1. Objects with threshold queries. Definition 5.1 extends Shapiro *et al.*'s definition of a state-based object with a threshold query method t:

Definition 5.1 (state-based object with threshold queries). A *state-based object with threshold queries* (henceforth *object*) is a tuple (S, s^0, q, t, u, m), where S is a set of states, $s^0 \in S$ is the initial state, q is a *query method*, t is a *threshold query method*, u is an *update method*, and m is a *merge method*.

In order to give a semantics to the threshold query method t, we need to formally define the notion of a threshold set. The notion of "threshold set" that I use here is the generalized formulation of threshold sets, based on *activation sets*, that I described previously in Section 2.6.2.

Definition 5.2 (threshold set). A *threshold set with respect to a lattice* $(S, \leq)$ is a set $\mathcal{S} = \{S_a, S_b, \dots\}$ of one or more sets of *activation states*, where each set of activation states is a subset of S, the set of lattice elements, and where the following *pairwise incompatibility* property holds:

For all $S_a, S_b \in \mathcal{S}$, if $S_a \neq S_b$, then for all activation states $s_a \in S_a$ and for all activation states $s_b \in S_b$, $s_a \sqcup s_b = \top$, where $\sqcup$ is the join operation induced by $\leq$ and $\top$ is the greatest element of $(S, \leq)$.

In our model, we assume a finite set of n processes $p_1, \ldots, p_n$, and consider a single replicated object with one replica at each process, with replica i at process p_i. Processes may crash silently; we say that a non-crashed process is *correct*.

Every replica has initial state s^0. Methods execute at individual replicas, possibly updating that replica's state. The kth method execution at replica i is written $f_i^k(a)$, where k is ≥ 1 and f is either q, t, u, or m, and a is the arguments to f, if any. Methods execute sequentially at each replica. The state of replica i after the kth method execution at i is s_i^k. We say that states s and s' are equivalent, written $s \equiv s'$, if $q(s) = q(s')$.

5.2.2. Causal histories. An object's *causal history* is a record of all the updates that have happened at all replicas. The causal history does not track the order in which updates happened, merely that they did happen. The *causal history at replica i after execution k* is the set of all updates that have happened at replica i after execution k. Definition 5.3 updates Shapiro *et al.*'s definition of causal history for a state-based object to account for t (a trivial change, since execution of t does not change a replica's causal history):

Definition 5.3 (causal history). A *causal history* is a sequence $[c_1, \ldots, c_n]$, where c_i is a set of the updates that have occurred at replica i. Each c_i is initially $\emptyset$. If the kth method execution at replica i is:

- a query q or a threshold query t, then the causal history at replica i after execution k does not change: $c_i^k = c_i^{k-1}$.
- an update $u_i^k(a)$, then the causal history at replica i after execution k is $c_i^k = c_i^{k-1} \cup u_i^k(a)$.
- a merge $m_i^k(s_{i'}^{k'})$, then the causal history at replica i after execution k is the union of the local and remote histories: $c_i^k = c_i^{k-1} \cup c_{i'}^{k'}$.

We say that an update is *delivered at replica i* if it is in the causal history at replica i.

5.2.3. Threshold CvRDTs and the semantics of blocking. With the previous definitions in place, we can give the definition of a CvRDT that supports threshold queries:

Definition 5.4 (CvRDT with threshold queries). A *convergent replicated data type with threshold queries* (henceforth *threshold CvRDT*) is an object equipped with a partial order $\leq$, written $(S, \leq, s^0, q, t, u, m)$, that has the following properties:

- S forms a join-semilattice ordered by $\leq$.
- S has a greatest element $\top$ according to $\leq$.
- The query method q takes no arguments and returns the local state.
- The threshold query method t takes a threshold set $\mathcal{S}$ as its argument, and has the following semantics: let $t_i^{k+1}(\mathcal{S})$ be the $k+1$th method execution at replica i, where $k \geq 0$. If, for some activation state s_a in some (unique) set of activation states $S_a \in \mathcal{S}$, the condition $s_a \leq s_i^k$ is met, $t_i^{k+1}(\mathcal{S})$ returns the set of activation states S_a. Otherwise, $t_i^{k+1}(\mathcal{S})$ returns the distinguished value block.
- The update method u takes a state as argument and updates the local state to it.
- State is *inflationary* across updates: if u updates a state s to s', then $s \leq s'$.
- The merge method m takes a remote state as its argument, computes the join of the remote state and the local state with respect to $\leq$, and updates the local state to the result.

and the q, t, u, and m methods have no side effects other than those listed above.

We use the block return value to model t's "blocking" behavior as a mathematical function with no intrinsic notion of running duration. When we say that a call to t "blocks", we mean that it immediately returns block, and when we say that a call to t "unblocks", we mean that it returns a set of activation states S_a.

Modeling blocking as a distinguished value introduces a new complication: we lose determinism, because a call to t at a particular replica may return either block or a set of activation states S_a, depending

on the replica's state at the time it is called. However, we can conceal this nondeterminism with an additional layer over the nondeterministic API exposed by t. This additional layer simply *polls* t, calling it repeatedly until it returns a value other than block. Calls to t at a replica that are made by this "polling layer" count as method executions at that replica, and are arbitrarily interleaved with other method executions at the replica, including updates and merges. The polling layer itself need not do any computation other than checking to see whether t returns block or something else; in particular, the polling layer does not need to compare activation states to replica states, since that comparison is done by t itself.

The set of activation states S_a that a call to t returns when it unblocks is unique because of the pairwise incompatibility property required of threshold sets: without it, different orderings of updates could allow the same threshold query to unblock in different ways, introducing nondeterminism that would be observable beyond the polling layer.

5.2.4. Threshold CvRDTs are strongly eventually consistent. We can define eventual consistency and strong eventual consistency exactly as Shapiro *et al*. do in their model. In the following definitions, a *correct replica* is a replica at a correct process, and the symbol $\Diamond$ means "eventually":

Definition 5.5 (eventual consistency (EC)). An object is *eventually consistent* (EC) if the following three conditions hold:

- *Eventual delivery*: An update delivered at some correct replica is eventually delivered at all correct replicas: $\forall i, j : f \in c_i \implies \Diamond f \in c_j$.
- *Convergence*: Correct replicas at which the same updates have been delivered eventually have equivalent state: $\forall i, j : c_i = c_j \implies \Diamond s_i \equiv s_j$.
- *Termination*: All method executions halt.

Definition 5.6 (strong eventual consistency (SEC)). An object is *strongly eventually consistent* (SEC) if it is eventually consistent and the following condition holds:

- *Strong convergence*: Correct replicas at which the same updates have been delivered have equivalent state: $\forall i, j : c_i = c_j \implies s_i \equiv s_j$.

Since we model blocking threshold queries with block, we need not be concerned with threshold queries not necessarily terminating. Determinism does *not* rule out queries that return block every time they are called (and would therefore cause the polling layer to block forever). However, we guarantee that if a threshold query returns block every time it is called during a complete run of the system, it will do so on *every* run of the system, regardless of scheduling. That is, it is not possible for a query to cause the polling layer to block forever on some runs, but not on others.

Finally, we can directly leverage Shapiro *et al.*'s SEC result for CvRDTs to show that a threshold CvRDT is SEC:

Theorem 5.1 (Strong Eventual Consistency of Threshold CvRDTs). *Assuming eventual delivery and termination, an object that meets the criteria for a threshold CvRDT is SEC.*

Proof. From Shapiro *et al.*, we have that an object that meets the criteria for a CvRDT is SEC [45]. Shapiro *et al.*'s proof also assumes that eventual delivery and termination hold for the object, and proves that strong convergence holds — that is, that given causal histories c_i and c_j for respective replicas i and j, that their states s_i and s_j are equivalent. The proof relies on the commutativity of the least upper bound operation. Since, according to our Definition 5.3, threshold queries do not affect causal history, we can leverage Shapiro *et al.*'s result to say that a threshold CvRDT is also SEC. □

5.3. Determinism of threshold queries

Neither eventual consistency nor strong eventual consistency imply that *intermediate* results of the same query q on different replicas of a threshold CvRDT will be deterministic. For deterministic intermediate results, we must use the threshold query method t. We can show that t is deterministic *without* requiring that the same updates have been delivered at the replicas in question at the time that t runs.

Theorem 5.2 establishes a determinism property for threshold queries of CvRDTs, porting the determinism result previously established for threshold reads for LVars to a distributed setting.

Theorem 5.2 (Determinism of Threshold Queries). *Suppose a given threshold query t on a given threshold CvRDT returns a set of activation states S_a when executed at a replica i. Then, assuming eventual delivery and that no replica's state is ever $\top$ at any point in the execution:*

(1) *t will always return S_a on subsequent executions at i, and*

(2) *t will* eventually *return S_a when executed at* any *replica, and will* block *until it does so.*

Proof. The proof relies on transitivity of $\leq$ and eventual delivery of updates; see Section A.18 for the complete proof. □

Although Theorem 5.2 must assume eventual delivery, it does *not* need to assume strong convergence or even ordinary convergence. It so happens that we have strong convergence as part of strong eventual consistency of threshold CvRDTs (by Theorem 5.1), but we do not need it to prove Theorem 5.2. In particular, there is no need for replicas to have the same state in order to return the same result from a particular threshold query. The replicas merely both need to be above an activation state from a unique set of activation states in the query's threshold set. Indeed, the replicas' states may in fact trigger *different* activation states from the same set of activation states.

Theorem 5.2's requirement that no replica's state is ever $\top$ rules out situations in which replicas disagree in a way that cannot be resolved normally. Recall from Section 2.4.1 that in the LVars model, when a program contains conflicting writes that would cause an LVar to reach its $\top$ state, a threshold read in that program *can* behave nondeterministically. However, since in our definition of observable determinism, only the final outcome of a program counts, this nondeterministic behavior of `get` in the presence of conflicting writes is not observable: such a program would always have **error** as its final outcome. In our setting of CvRDTs, though, we do not have a notion of "program", nor of the final outcome

thereof. Rather than having to define those things and then define a notion of observable determinism based on them, I rule out this situation by assuming that no replica's state goes to $\top$.

CHAPTER 6

Related work

Work on deterministic parallel programming models is long-standing. As we have seen, what deterministic parallel programming models have in common is that they all must do something to restrict access to mutable state shared among concurrent computations so that schedule nondeterminism cannot be observed. Depending on the model, restricting access to shared mutable state might involve disallowing sharing entirely [41], only allowing single assignments to shared references [51, 3, 11], allowing sharing only by a limited form of message passing [29], ensuring that concurrent accesses to shared state are disjoint [8], resolving conflicting updates after the fact [32], or some combination of these approaches. These constraints can be imposed at the language or API level, within a type system or at runtime.

In particular, the LVars model was inspired by two traditional deterministic parallel programming models based on monotonically-growing shared data structures: first, Kahn process networks [29], in which a network of processes communicate with each other through blocking FIFO channels with ever-growing channel histories; and, second, IVars [3], single-assignment locations with blocking read semantics.

LVars are general enough to subsume both IVars and KPNs: a lattice of channel histories with a prefix ordering allows LVars to represent FIFO channels that implement a Kahn process network, whereas instantiating λ_{LVar} with a lattice with one "empty" state and multiple "full" states (where $\forall i.\ \mathit{empty} < \mathit{full}_i$) results in a parallel single-assignment language with a store of IVars, as we saw in Chapter 2. Hence LVars provide a framework for generalizing and unifying these two existing approaches to deterministic parallelism. In this chapter, I describe some more recent contributions to the literature, and how the LVars model relates to them.

6.1. Deterministic Parallel Java (DPJ)

DPJ [8, 7] is a deterministic language consisting of a system of annotations for Java code. A sophisticated region-based type system ensures that a mutable region of the heap is, essentially, passed linearly to an exclusive writer, thereby ensuring that the state accessed by concurrent threads is disjoint. DPJ does, however, provide a way to unsafely assert that operations commute with one another (using the `commuteswith` form) to enable concurrent mutation.

The LVars model differs from DPJ in that it allows overlapping shared state between threads as the default. Moreover, since LVar effects are already commutative, we avoid the need for `commuteswith` annotations. Finally, it is worth noting that while in DPJ, commutativity annotations have to appear in application-level code, in LVish only the data-structure author needs to write trusted code. The application programmer can run untrusted code that still enjoys a (quasi-)determinism guarantee, because only (quasi-)deterministic programs can be expressed as LVish `Par` computations.

More recently, Bocchino *et al*. [9] proposed a type and effect system that allows for the incorporation of nondeterministic sections of code in DPJ. The goal here is different from ours: while they aim to support *intentionally* nondeterministic computations such as those arising from optimization problems like branch-and-bound search, the quasi-determinism in LVish arises as a result of schedule nondeterminism.

6.2. FlowPools

Prokopec *et al*. [42] propose a data structure with an API closely related to LVars extended with freezing and handlers: a FlowPool is a bag (that is, a multiset) that allows concurrent insertions but forbids removals, a `seal` operation that forbids further updates, and combinators like `foreach` that invoke callbacks as data arrives in the pool. To retain determinism, the `seal` operation requires explicitly passing the expected bag *size* as an argument, and the program will raise an exception if the bag goes over the expected size.

While this interface has a flavor similar to that of LVars, it lacks the ability to detect quiescence, which is crucial for expressing algorithms like graph traversal, and the `seal` operation is awkward to use when the structure of data is not known in advance. By contrast, the `freeze` operation on LVars does not require such advance knowledge, but moves the model into the realm of quasi-determinism. Another important difference is the fact that LVars are *data structure-generic*: both our formalism and our library support an unlimited collection of data structures, whereas FlowPools are specialized to bags.

6.3. Concurrent Revisions

Burckhardt *et al*. [12] propose a formalism for eventual consistency based on graphs called *revision diagrams*, and Leijen, Burckhardt, and Fahndrich apply the revision diagrams approach to guaranteed-deterministic concurrent functional programming [32]. Their *Concurrent Revisions* (CR) programming model uses isolation types to distinguish regions of the heap shared by multiple mutators. Rather than enforcing exclusive access in the style of DPJ, CR clones a copy of the state for each mutator, using a deterministic "merge function" for resolving conflicts in local copies at join points.

In CR, variables can be annotated as being shared between a "joiner" thread and a "joinee" thread. Unlike the least-upper-bound writes of LVars, CR merge functions are *not* necessarily commutative; indeed, the default CR merge function is "joiner wins". Determinism is enforced by the programming model allowing the programmer to specify which of two writing threads should prevail, regardless of the order in which those writes arrive, and the states that a shared variable can take on need not form a lattice. Still, semilattices turn up in the metatheory of CR: in particular, Burckhardt and Leijen [13] show that revision diagrams are semilattices, and that therefore, for any two vertices in a CR revision diagram, there exists a *greatest common ancestor* state that can be used to determine what changes each side has made—an interesting duality with the LVars model (in which any two LVar states have a least upper bound).

Although versioned variables in CR could model lattice-based data structures—if they used least upper bound as their merge function for conflicts—the programming model nevertheless differs from the LVars model in that effects only become visible at the end of parallel regions, as opposed to the asynchronous communication within parallel regions that the LVars model allows. This semantics precludes the use of traditional lock-free data structures for representing versioned variables.

6.4. Conflict-free replicated data types

In Chapter 5, I presented a way to equip lattice-based distributed data structures with LVar-style threshold reads, resulting in a way to make deterministic *threshold queries* of them. My is based on Shapiro *et al.*'s work on conflict-free replicated data types (CRDTs) [**45**, **44**] and in particular their work on the lattice-based formulation of CRDTs, called *convergent replicated data types* or CvRDTs, which I discuss in detail in Section 5.1.

As we saw in Chapter 5, database services such as Amazon's SimpleDB [**53**] allow for both eventually consistent and strongly consistent reads, chosen at a per-query granularity. Terry *et al.*'s Pileus key-value store [**50**] takes the idea of mixing consistency levels further: instead of requiring the application developer to choose the consistency level of a particular query at development time, the system allows the developer to specify a service-level agreement that can dynamically adapt to changing network conditions, for instance. Hence choosing consistency at a per-query level and mixing consistency levels within a single application is not a new idea. Rather, the new contribution that I make by adding threshold queries to CvRDTs is to establish lattice-based data structures as a unifying formal foundation for both eventually consistent and strongly consistent queries.

6.5. Bloom and Bloom^L

Other authors have also used lattices as a framework for establishing formal guarantees about eventually consistent systems and distributed programs. The Bloom language for distributed database programming guarantees eventual consistency for distributed data collections that are updated monotonically. The initial formulation of Bloom [2] had a notion of monotonicity based on set inclusion, which is analogous to the store ordering relation in the (IVar-based) Featherweight CnC system that I described in Section 2.3.4. Later, Conway *et al*. [16] generalized Bloom to a more flexible lattice-parameterized system, Bloom^L, in a manner analogous to the generalization from IVars to LVars. Bloom^L combines ideas from the aforementioned work on CRDTs [45, 44] with *monotonic logic*, resulting in a lattice-parameterized, confluent language that is a close (but independently invented) relative to the LVars model. Bloom(^{L}) is implemented as a domain-specific language embedded in Ruby, and a monotonicity analysis pass rules out programs that would perform non-monotonic operations on distributed data collections (such as the removal of elements from a set). By contrast, in the LVars model (and in the LVish library), monotonicity is enforced by the API presented by LVars, and since the LVish library is implemented in Haskell, we can rely on Haskell's type system for fine-grained effect tracking and monadic encapsulation of LVar effects.

LK: Contemplating adding another section of related work here, about separation logic, frame properties, fictional separation logic, and Views...

CHAPTER 7

Summary and future work

As single-assignment languages and Kahn process networks demonstrate, monotonicity can serve as the foundation of diverse deterministic-by-construction parallel programming models. The LVars programming model takes monotonicity as a starting point and generalizes single assignment to monotonic multiple assignment, parameterized by a lattice. The LVars model, and the accompanying LVish library, support my claim that lattice-based data structures are a general and practical foundation for deterministic and quasi-deterministic parallel and distributed programming.

7.1. Remapping the deterministic parallel landscape

Let us reconsider how LVars fit into the deterministic parallel programming landscape that we mapped out in Chapter 1:

- *No-shared-state parallelism:* The purely functional core of the λ_{LVar} and λ_{LVish} calculi (and of the LVish Haskell library) allow no-shared-state, pure task parallelism. Of course, shared-state programming is the point of the LVars model. However, it is significant that we take pure programming as a starting point, because it distinguishes the LVars model from approaches such as DPJ [8] that begin with a parallel (but nondeterministic) language and then restrict the sharing of state to regain determinism. The LVars model works in the other direction: it begins with a deterministic parallel language without shared state, and then adds limited effects that retain determinism.
- *Data-flow parallelism:* As we have seen, because LVars are lattice-generic, the LVars model can subsume Kahn process networks and other parallel programming models based on data flow, since we can use LVars to represent a lattice of channel histories, ordered by a prefix ordering.

- *Single-assignment parallelism:* Single-assignment variables, or IVars, are also subsumed by LVars: an IVar is an LVar whose lattice has one "empty" state and multiple "full" states (where $\forall i.\ \mathit{empty} < \mathit{full}_i$). In fact, given how useful IVars are, the subsumption of IVars by LVars demonstrates that immutability is an important special case of monotonicity.[1]
- *Imperative disjoint parallelism:* Although the LVars model generally does *not* require that the state accessed by concurrent threads is disjoint, this style of ensuring determinism is still compatible with the LVars model, and it is practically achievable using the `ParST` monad transformer in LVish, as we saw in Section 4.3.

In addition to subsuming or accommodating all these existing points on the landscape, the LVars model identifies a new point on the map: a *quasi-deterministic* programming model, allowing programs that perform *freezing* and are deterministic modulo exceptions. The ability to freeze and read the exact contents of an LVar greatly increases the expressiveness of the LVars model, especially when used in conjunction with event handlers. Furthermore, we can regain full determinism by ensuring that freezing happens last, and, as we saw in Section 4.2.6, it is possible to enforce this "freeze after writing" requirement at the implementation level.

Of course, there is still more work to do. For example, although imperative disjoint parallelism and the LVars model seem to be compatible, as evidenced by the use of `ParST` in LVish, we have not yet formalized their relationship. In fact, this is an example of a general pattern in which the LVish library is usually one step ahead of the LVars formalism: to take another example, the LVish library supported the general inflationary, commutative writes of Section 2.6.1 well before the notion had been formalized in λ_{LVish}. Moreover, even for the parts of the LVish library that *are* fully accounted for in the model, we do not have proof that the LVish library is a faithful implementation of the formal LVars model.

[1] As Neil Conway puts it, "Immutability is a special case of monotone growth, albeit a particularly useful one" (https://twitter.com/neil_conway/status/392337034896871424).

Although it is unlikely that this game of catch-up can ever be won, an interesting direction to pursue for future work would be a *verified* implementation of LVish, for instance, in a dependently typed programming language. Even though a fully verified implementation of LVish (including the scheduler implementation) might be unrealistic, a more manageable might be invidividiually verified LVar data structures. For example, in a dependently typed language such as Idris or Agda, we could use the type system to express properties that must be true of an LVar, such as that the states that it can take on form a lattice and that writes are commutative and inflationary.

7.2. Distributed programming and the future of LVars and LVish

LK: Maybe some of this material should actually be moved to the end of Chapter 5, because that chapter ends sort of abruptly. I'm not sure.

Most of this dissertation concerns the problem of how to program *parallel* systems, in which programs are running on multiple processors. However, I am also concerned with the problem of how to program *distributed* systems, in which programs must run on networked computers around the world. Enormous bodies of work have been developed to deal with both of these problems, and one of the roles that programming languages research can play is to try to find unifying abstractions between the two. It is in that spirit that I have explored the relationship of LVars to existing work on distributed systems.

This work is made much easier by the fact that LVars have a close cousin in the distributed systems literature in convergent replicated data types (CvRDTs) [**45**, **44**], which leverage lattice properties to guarantee that all replicas of an object (for instance, in a distributed database) are eventually consistent. Chapter 5 begins to explore the relationship between LVars and CvRDTs by porting LVar-style threshold reads to the CvRDT setting. However, there is *much* more work to do here. Most immediately, although the idea of a single lattice-based framework for reasoning about both strongly consistent and eventually consistent queries of distributed data is appealing and elegant, it is not yet clear what the applications for threshold-readable CvRDTs are.

Second, it should also be possible to back-port ideas from the realm of CvRDTs to LVars. We have taken a few steps in this direction—in fact, supporting the aforementioned commutative and inflationary updates is actually a step toward making LVars more like CvRDTs, since CvRDTs have always allowed arbitrary inflationary and commutative writes to individual replicas (the least-upper-bound operation is only used when replicas' states are *merged* with one another). But the LVars model could further benefit from applying techniques pioneered by CvRDTs to support data structures that allow seemingly non-monotonic updates, such as counters that support decrements as well as increments and sets that support removals as well as additions.

Finally, a huge remaining difference between LVars and CvRDTs is that in the LVars model, we do not have to contend with replication! The LVars model is a shared-memory model, and when an LVar is updated, all reading threads can immediately see the update. CvRDTs, as well as distributed lattice-based programming languages like Bloom [2, 16], may serve as a source of inspiration for a future version of LVish that supports distributed execution.

LK: Saying something about Adam's work on meta-par would make sense here.

LK: Maybe say something about my eventual goal of connecting up LVars, CvRDTs, and separation logic?

Bibliography

[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`. 14, 16

[2] P. Alvaro, N. Conway, J. Hellerstein, and W. R. Marczak. Consistency analysis in Bloom: a CALM and collected approach. In *CIDR*, 2011. 135, 140

[3] Arvind, R. S. Nikhil, and K. K. Pingali. I-structures: data structures for parallel computing. *ACM Trans. Program. Lang. Syst.*, 11(4), Oct. 1989. 3, 12, 17, 22, 131

[4] H. Attiya, R. Guerraoui, D. Hendler, P. Kuznetsov, M. M. Michael, and M. Vechev. Laws of order: expensive synchronization in concurrent algorithms cannot be eliminated. In *POPL*, 2011. 81

[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. 3

[6] G. E. Blelloch, J. T. Fineman, P. B. Gibbons, and J. Shun. Internally deterministic parallel algorithms can be fast. In *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, PPoPP '12. ACM, New York, NY, USA, 2012. 35

[7] R. L. Bocchino, Jr., V. S. Adve, S. V. Adve, and M. Snir. Parallel programming must be deterministic by default. In *HotPar*, 2009. 3, 11, 132

[8] R. L. Bocchino, Jr., V. S. Adve, D. Dig, S. V. Adve, S. Heumann, R. Komuravelli, J. Overbey, P. Simmons, H. Sung, and M. Vakilian. A type and effect system for deterministic parallel Java. In *OOPSLA*, 2009. 3, 11, 80, 95, 131, 132, 137

[9] R. L. Bocchino, Jr. et al. Safe nondeterminism in a deterministic-by-default parallel language. In *POPL*, 2011. 132

[10] E. Brewer. CAP twelve years later: How the "rules" have changed. `http://www.infoq.com/articles/cap-twelve-years-later-how-the-rules-have-changed`, 2012. 119

[11] 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(3-4), Aug. 2010. 3, 12, 17, 26, 31, 40, 131

[12] S. Burckhardt, M. Fahndrich, D. Leijen, and M. Sagiv. Eventually consistent transactions. In *ESOP*, 2012. 133

[13] S. Burckhardt and D. Leijen. Semantics of concurrent revisions. In *ESOP*, 2011. 133

[14] M. M. T. Chakravarty, R. Leshchinskiy, S. Peyton Jones, G. Keller, and S. Marlow. Data Parallel Haskell: A status report. In *DAMP*, 2007. 2

[15] D. Chase and Y. Lev. Dynamic circular work-stealing deque. In *SPAA*, 2005. 106

[16] N. Conway, W. Marczak, P. Alvaro, J. M. Hellerstein, and D. Maier. Logic and lattices for distributed programming. In *SOCC*, 2012. 135, 140

[17] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *SOSP*, 2007. 119, 120

[18] C. Earl, I. Sergey, M. Might, and D. Van Horn. Introspective pushdown analysis of higher-order programs. In *ICFP*, 2012. 112

[19] F. Ellen, Y. Lev, V. Luchangco, and M. Moir. SNZI: Scalable NonZero Indicators. In *PODC*, 2007. 105

[20] M. Felleisen, R. B. Findler, and M. Flatt. *Semantics Engineering with PLT Redex*. The MIT Press, 1st edition, 2009. 9, 207

[21] J. Felsenstein. PHYLIP - Phylogeny Inference Package (Version 3.2). *Cladistics*, 5:164–166, 1989. 115

[22] M. Fluet, M. Rainey, J. Reppy, and A. Shaw. Implicitly-threaded parallelism in Manticore. In *ICFP*, 2008. 30

[23] K. Fraser. *Practical lock-freedom*. PhD thesis, 2004. 104

[24] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the Cilk-5 multithreaded language. In *PLDI*, 1998. 11

[25] S. Gilbert and N. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2), June 2002. 119

[26] 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. 3, 12

[27] S. Herhut, R. L. Hudson, T. Shpeisman, and J. Sreeram. River trail: A path to parallelism in javascript. In *OOPSLA*, 2013. 2

[28] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2008. 81, 112

[29] G. Kahn. The semantics of a simple language for parallel programming. In J. L. Rosenfeld, editor, *Information processing*. North Holland, Amsterdam, Aug. 1974. 2, 12, 131

[30] O. Kiselyov, A. Sabry, and C. Swords. Extensible effects: an alternative to monad transformers. In *Haskell*, 2013. 85

[31] E. Lee and D. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, 1987. 2

[32] D. Leijen, M. Fahndrich, and S. Burckhardt. Prettier concurrency: purely functional concurrent revisions. In *Haskell*, 2011. 131, 133

[33] S. Marlow. *Parallel and Concurrent Programming in Haskell*. O'Reilly Media, 2013. 86

[34] S. Marlow, P. Maier, H.-W. Loidl, M. K. Aswad, and P. Trinder. Seq no more: better strategies for parallel Haskell. In *Haskell*, 2010. 2, 11, 109

[35] S. Marlow, R. Newton, and S. Peyton Jones. A monad for deterministic parallelism. In *Haskell*, 2011. 3, 7, 11, 12, 16, 17, 79, 80

[36] S. Marlow, S. Peyton Jones, and S. Singh. Runtime support for multicore Haskell. In *ICFP*, 2009. 94

[37] M. M. Michael, M. T. Vechev, and V. A. Saraswat. Idempotent work stealing. In *PPoPP*, 2009. 106

[38] M. Might. k-CFA: Determining types and/or control-flow in languages like Python, Java and Scheme. http://matt.might.net/articles/implementation-of-kcfa-and-0cfa/. 108

[39] R. R. Newton and I. L. Newton. PhyBin: binning trees by topology. *PeerJ*, 1:e187, Oct. 2013. 8, 115

[40] R. S. Nikhil. Id language reference manual, 1991. 17

[41] S. L. Peyton Jones, R. Leshchinskiy, G. Keller, and M. M. T. Chakravarty. Harnessing the multicores: Nested data parallelism in Haskell. In *FSTTCS*, 2008. 2, 131

[42] A. Prokopec, H. Miller, T. Schlatter, P. Haller, and M. Odersky. FlowPools: a lock-free deterministic concurrent dataflow abstraction. In *LCPC*, 2012. 132

[43] J. H. Reppy. *Concurrent Programming in ML*. Cambridge University Press, Cambridge, England, 1999. 3

[44] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. A comprehensive study of Convergent and Commutative Replicated Data Types. Technical Report RR-7506, INRIA, Jan. 2011. 8, 120, 121, 134, 135, 139

[45] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. Conflict-free replicated data types. In *SSS*, 2011. 8, 87, 120, 121, 123, 128, 134, 135, 139

[46] 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. 30

[47] J. Sukumaran and M. T. Holder. DendroPy: a Python library for phylogenetic computing. *Bioinformatics*, 26:1569–1571, 2010. 115

[48] S.-J. Sul and T. L. Williams. A randomized algorithm for comparing sets of phylogenetic trees. In *APBC*, 2007. 8, 115

[49] D. Terei, D. Mazières, S. Marlow, and S. Peyton Jones. Safe haskell. In *Haskell*, 2012. 11

[50] D. B. Terry, V. Prabhakaran, R. Kotla, M. Balakrishnan, M. K. Aguilera, and H. Abu-Libdeh. Consistency-based service level agreements for cloud storage. In *SOSP*, 2013. 120, 134

[51] L. G. Tesler and H. J. Enea. A language design for concurrent processes. In *AFIPS*, 1968 (Spring). 12, 17, 131

[52] W. Vogels. Eventually consistent. *Commun. ACM*, 52(1), Jan. 2009. 8, 119

[53] W. Vogels. Choosing consistency. http://www.allthingsdistributed.com/2010/02/strong_consistency_simpledb.html, 2010. 120, 134

APPENDIX A

Proofs

A.1. Proof of Lemma 2.1

Proof. Consider an arbitrary permutation π. For part 1, we have to show that if $\sigma \longhookrightarrow \sigma'$ then $\pi(\sigma) \longhookrightarrow \pi(\sigma')$, and that if $\pi(\sigma) \longhookrightarrow \pi(\sigma')$ then $\sigma \longhookrightarrow \sigma'$.

For the forward direction of part 1, suppose $\sigma \longhookrightarrow \sigma'$. We have to show that $\pi(\sigma) \longhookrightarrow \pi(\sigma')$. We proceed by cases on the rule by which σ steps to σ'.

- Case E-Beta: $\sigma = \langle S;\ (\lambda x.\, e)\ v\rangle$, and $\sigma' = \langle S;\ e[x := v]\rangle$.

  To show: $\pi(\langle S;\ (\lambda x.\, e)\ v\rangle) \longhookrightarrow \pi(\langle S;\ e[x := v]\rangle)$.

  By Definition 2.9, $\pi(\sigma) = \langle \pi(S);\ \pi(\lambda x.\, e)\, \pi(v)\rangle$, which, by Definition 2.7, is equal to $\langle \pi(S);\ (\lambda x.\, \pi(e))\, \pi(v)\rangle$.

  By E-Beta, $\langle \pi(S);\ (\lambda x.\, \pi(e))\, \pi(v)\rangle$ steps to $\langle \pi(S);\ \pi(e)[x := \pi(v)]\rangle$.

  By Definition 2.7, $\langle \pi(S);\ \pi(e)[x := \pi(v)]\rangle$ is equal to $\langle \pi(S);\ \pi(e[x := v])\rangle$.

  Hence $\langle \pi(S);\ (\lambda x.\, \pi(e))\ \pi(v)\rangle$ steps to $\langle \pi(S);\ \pi(e[x := v])\rangle$, which is equal to $\pi(\langle S;\ e[x := v]\rangle)$ by Definition 2.9. Hence the case is satisfied.

- Case E-New: $\sigma = \langle S;\ \texttt{new}\rangle$, and $\sigma' = \langle S[l \mapsto \bot];\ l\rangle$.

  To show: $\pi(\langle S;\ \texttt{new}\rangle) \longhookrightarrow \pi(\langle S[l \mapsto \bot];\ l\rangle)$.

  By Definition 2.9, $\pi(\sigma) = \langle \pi(S);\ \pi(\texttt{new})\rangle$, which, by Definition 2.7, is equal to $\langle \pi(S);\ \texttt{new}\rangle$.

  By E-New, $\langle \pi(S);\ \texttt{new}\rangle$ steps to $\langle (\pi(S))[l' \mapsto \bot];\ l'\rangle$, where $l' \notin \mathit{dom}(\pi(S))$.

  It remains to show that $\langle (\pi(S))[l' \mapsto \bot];\ l'\rangle$ is equal to $\pi(\langle S[l \mapsto \bot];\ l\rangle)$.

  By Definition 2.9, $\pi(\langle S[l \mapsto \bot];\ l\rangle)$ is equal to $\langle \pi(S[l \mapsto \bot]);\ \pi(l)\rangle$, which is equal to $\langle (\pi(S))[\pi(l) \mapsto \bot];\ \pi(l)\rangle$.

So, we have to show that $\langle(\pi(S))[l' \mapsto \bot];\ l'\rangle$ is equal to $\langle(\pi(S))[\pi(l) \mapsto \bot];\ \pi(l)\rangle$. Since we know (from the side condition of E-New) that $l \notin \mathit{dom}(S)$, it follows that $\pi(l) \notin \pi(\mathit{dom}(S))$. Therefore, in $\langle(\pi(S))[l' \mapsto \bot];\ l'\rangle$, we can α-rename l' to $\pi(l)$, and so the two configurations are equal and the case is satisfied.

- Case E-Put: $\sigma = \langle S;\ \texttt{put}\ l\ d_2\rangle$, and $\sigma' = \langle S[l \mapsto d_1 \sqcup d_2];\ ()\rangle$.

  To show: $\pi(\langle S;\ \texttt{put}\ l\ d_2\rangle) \longrightarrow \pi(\langle S[l \mapsto d_1 \sqcup d_2];\ ()\rangle)$.

  By Definition 2.9, $\pi(\sigma) = \langle\pi(S);\ \texttt{put}\ \pi(l)\,\pi(d_2)\rangle$, which, by Definition 2.7, is equal to $\langle\pi(S);\ \texttt{put}\ \pi(l)\,d_2\rangle$.

  By E-Put, $\langle\pi(S);\ \texttt{put}\ \pi(l)\,d_2\rangle$ steps to $\langle(\pi(S))[\pi(l) \mapsto d_1 \sqcup d_2];\ ()\rangle$, since $S(l) = (\pi(S))(\pi(l)) = d_1$.

  It remains to show that $\langle(\pi(S))[\pi(l) \mapsto d_1 \sqcup d_2];\ ()\rangle$ is equal to $\pi(\langle S[l \mapsto d_1 \sqcup d_2];\ ()\rangle)$.

  By Definition 2.9, $\pi(\langle S[l \mapsto d_1 \sqcup d_2];\ ()\rangle)$ is equal to $\langle(\pi(S))[\pi(l) \mapsto \pi(d_1 \sqcup d_2)];\ \pi(())\rangle$, which, by Definition 2.7, is equal to $\langle(\pi(S))[\pi(l) \mapsto d_1 \sqcup d_2];\ ()\rangle$, and so the two configurations are equal and the case is satisfied.

- Case E-Put-Err: $\sigma = \langle S;\ \texttt{put}\ l\ d_2\rangle$, and $\sigma' = \textbf{error}$.

  To show: $\pi(\langle S;\ \texttt{put}\ l\ d_2\rangle) \longrightarrow \pi(\textbf{error})$.

  By Definition 2.9, $\pi(\sigma) = \langle\pi(S);\ \texttt{put}\ \pi(l)\,\pi(d_2)\rangle$, which, by Definition 2.7, is equal to $\langle\pi(S);\ \texttt{put}\ \pi(l)\,d_2\rangle$.

  By E-Put-Err, $\langle\pi(S);\ \texttt{put}\ \pi(l)\,d_2\rangle$ steps to **error**, since $S(l) = (\pi(S))(\pi(l)) = d_1$.

  Since $\pi(\textbf{error}) = \textbf{error}$ by Definition 2.9, the case is complete.

- Case E-Get: $\sigma = \langle S;\ \texttt{get}\ l\ T\rangle$, and $\sigma' = \langle S;\ d_2\rangle$.

  To show: $\pi(\langle S;\ \texttt{get}\ l\ T\rangle) \longrightarrow \pi(\langle S;\ d_2\rangle)$.

  By Definition 2.9, $\pi(\sigma) = \langle\pi(S);\ \texttt{get}\ \pi(l)\,\pi(T)\rangle$, which, by Definition 2.7, is equal to $\langle\pi(S);\ \texttt{get}\ \pi(l)\,T\rangle$.

  By E-Get, $\langle\pi(S);\ \texttt{get}\ \pi(l)\,T\rangle$ steps to $\langle\pi(S);\ d_2\rangle$.

  By Definition 2.9, $\pi(\langle S;\ d_2\rangle) = \langle\pi(S);\ \pi(d_2)\rangle$, which, by Definition 2.7, is equivalent to $\langle\pi(S);\ d_2\rangle$.

  Therefore the case is complete.

For the reverse direction of part 1, suppose $\pi(\sigma) \longrightarrow \pi(\sigma')$. We have to show that $\sigma \longrightarrow \sigma'$.

We know from the forward direction of the proof that for all configurations σ and σ' and permutations π, if $\sigma \longhookrightarrow \sigma'$ then $\pi(\sigma) \longhookrightarrow \pi(\sigma')$. Hence since $\pi(\sigma) \longhookrightarrow \pi(\sigma')$, and since π^{-1} is also a permutation, we have that $\pi^{-1}(\pi(\sigma)) \longhookrightarrow \pi^{-1}(\pi(\sigma'))$. Since $\pi^{-1}(\pi(l)) = l$ for every $l \in \mathit{Loc}$, and that property lifts to configurations as well, we have that $\sigma \longhookrightarrow \sigma'$.

LK: Is the above enough of a proof?

For the forward direction of part 2, suppose $\sigma \longmapsto \sigma'$. We have to show that $\pi(\sigma) \longmapsto \pi(\sigma')$.

By inspection of the operational semantics, σ must be of the form $\langle S;\ E[e]\rangle$, and σ' must be of the form $\langle S';\ E[e']\rangle$. Hence we have to show that $\pi(\langle S;\ E[e]\rangle) \longmapsto \pi(\langle S';\ E[e']\rangle)$.

By Definition 2.9, $\pi(\langle S;\ E[e]\rangle)$ is equal to $\langle \pi(S);\ \pi(E[e])\rangle$, and $\pi(\langle S';\ E[e']\rangle)$ is equal to $\langle \pi(S');\ \pi(E[e'])\rangle$.

Furthermore, $\langle \pi(S);\ \pi(E[e])\rangle$ is equal to $\langle \pi(S);\ (\pi(E))[\pi(e)]\rangle$ and $\langle \pi(S');\ \pi(E[e'])\rangle$ is equal to $\langle \pi(S');\ (\pi(E))[\pi(e'$

So we have to show that $\langle \pi(S);\ (\pi(E))[\pi(e)]\rangle \longmapsto \langle \pi(S');\ (\pi(E))[\pi(e')]\rangle$.

From the premise of E-Eval-Ctxt, $\langle S;\ e\rangle \longhookrightarrow \langle S';\ e'\rangle$. Hence, by part 1, $\pi(\langle S;\ e\rangle) \longhookrightarrow \pi(\langle S';\ e'\rangle)$.

By Definition 2.9, $\pi(\langle S;\ e\rangle)$ is equal to $\langle \pi(S);\ \pi(e)\rangle$ and $\pi(\langle S';\ e'\rangle)$ is equal to $\langle \pi(S');\ \pi(e')\rangle$.

Hence $\langle \pi(S);\ \pi(e)\rangle \longhookrightarrow \langle \pi(S');\ \pi(e')\rangle$. Therefore, by E-Eval-Ctxt, $\langle \pi(S);\ E[\pi(e)]\rangle \longmapsto \langle \pi(S');\ E[\pi(e')]\rangle$ for all evaluation contexts E.

In particular, it is true that $\langle \pi(S);\ (\pi(E))[\pi(e)]\rangle \longmapsto \langle \pi(S');\ (\pi(E))[\pi(e')]\rangle$, as we were required to show.

For the reverse direction of part 2, suppose $\pi(\sigma) \longmapsto \pi(\sigma')$. We have to show that $\sigma \longmapsto \sigma'$.

We know from the forward direction of the proof that for all configurations σ and σ' and permutations π, if $\sigma \longmapsto \sigma'$ then $\pi(\sigma) \longmapsto \pi(\sigma')$. Hence since $\pi(\sigma) \longmapsto \pi(\sigma')$, and since π^{-1} is also a permutation, we have that $\pi^{-1}(\pi(\sigma)) \longmapsto \pi^{-1}(\pi(\sigma'))$. Since $\pi^{-1}(\pi(l)) = l$ for every $l \in \mathit{Loc}$, and that property lifts to configurations as well, we have that $\sigma \longmapsto \sigma'$.

LK: Is the above enough of a proof? □

A.2. Proof of Lemma 2.2

Proof. The proof is by cases on the rule by which σ steps to σ'.

- Case E-Beta:

  Given: $\langle S;\ (\lambda x.\, e)\ v\rangle \longhookrightarrow \langle S;\ e[x := v]\rangle$, and $\langle S;\ (\lambda x.\, e)\ v\rangle \longhookrightarrow \sigma''$.

  To show: There exists a π such that $\langle S;\ e[x := v]\rangle = \pi(\sigma'')$.

  By inspection of the operational semantics, the only reduction rule by which $\langle S;\ (\lambda x.\, e)\ v\rangle$ can step is E-Beta. Hence $\sigma'' = \langle S;\ e[x := v]\rangle$, and the case is satisfied by choosing π to be the identity function.

- Case E-New:

  Given: $\langle S;\ \texttt{new}\rangle \longhookrightarrow \langle S[l \mapsto \bot];\ l\rangle$, and $\langle S;\ \texttt{new}\rangle \longhookrightarrow \sigma''$.

  To show: There exists a π such that $\langle S;\ \texttt{new}\rangle = \pi(\sigma'')$.

  By inspection of the operational semantics, the only reduction rule by which $\langle S;\ \texttt{new}\rangle$ can step is E-New. Hence $\sigma'' = \langle S[l' \mapsto \bot];\ l'\rangle$. Since, by the side condition of E-New, neither l nor l' occur in $\mathit{dom}(S)$, the case is satisfied by choosing π to be the permutation that maps l' to l and is the identity on every other element of *Loc*.

- Case E-Put:

  Given: $\langle S;\ \texttt{put}\ l\ d_2\rangle \longhookrightarrow \langle S[l \mapsto d_1 \sqcup d_2];\ \texttt{()}\rangle$, and $\langle S;\ \texttt{put}\ l\ d_2\rangle \longhookrightarrow \sigma''$.

  To show: There exists a π such that $\langle S;\ \texttt{put}\ l\ d_2\rangle = \pi(\sigma'')$.

  By inspection of the operational semantics, and since $d_1 \sqcup d_2 \neq \top$ (from the premise of E-Put), the only reduction rule by which $\langle S;\ \texttt{put}\ l\ d_2\rangle$ can step is E-Put. Hence $\sigma'' = \langle S[l \mapsto d_1 \sqcup d_2];\ \texttt{()}\rangle$, and the case is satisfied by choosing π to be the identity function.

- Case E-Put-Err:

  Given: $\langle S;\ \texttt{put}\ l\ d_2\rangle \longhookrightarrow$ **error**, and $\langle S;\ \texttt{put}\ l\ d_2\rangle \longhookrightarrow \sigma''$.

  To show: There exists a π such that $\langle S;\ \texttt{put}\ l\ d_2\rangle = \pi(\sigma'')$.

By inspection of the operational semantics, and since $d_1 \sqcup d_2 = \top$ (from the premise of E-Put-Err), the only reduction rule by which $\langle S;\ \texttt{put}\ l\ d_2 \rangle$ can step is E-Put-Err. Hence $\sigma'' = \textbf{error}$, and the case is satisfied by choosing π to be the identity function.

- Case E-Get:

  Given: $\langle S;\ \texttt{get}\ l\ T \rangle \longrightarrow \langle S;\ d_2 \rangle$, and $\langle S;\ \texttt{get}\ l\ T \rangle \longrightarrow \sigma''$.

  To show: There exists a π such that $\langle S;\ \texttt{get}\ l\ T \rangle = \pi(\sigma'')$.

  By inspection of the operational semantics, the only reduction rule by which $\langle S;\ \texttt{put}\ l\ d_2 \rangle$ can step is E-Get. Hence $\sigma'' = \langle S;\ d_2 \rangle$, and the case is satisfied by choosing π to be the identity function.

□

A.3. Proof of Lemma 2.3

Proof. TODO: Prove this. □

A.4. Proof of Lemma 2.4

Proof. Suppose $\langle S;\ e \rangle \longrightarrow \langle S';\ e' \rangle$. We are required to show that $S \sqsubseteq_S S'$. The proof is by cases on the rule by which $\langle S;\ e \rangle$ steps to $\langle S';\ e' \rangle$.

- Case E-Beta:

  Immediate by the definition of $\sqsubseteq_S$, since S does not change.

- Case E-New:

  Given: $\langle S;\ \texttt{new} \rangle \longrightarrow \langle S[l \mapsto \bot];\ l \rangle$.

  To show: $S \sqsubseteq_S S[l \mapsto \bot]$.

  By Definition 2.2, we have to show that $\mathit{dom}(S) \subseteq \mathit{dom}(S[l \mapsto \bot])$ and that for all $l' \in \mathit{dom}(S)$, $S(l') \sqsubseteq (S[l \mapsto \bot])(l')$.

  By definition, a store update operation on S can only either update an existing binding in S or extend S with a new binding. Hence $\mathit{dom}(S) \subseteq \mathit{dom}(S[l \mapsto \bot])$.

From the side condition of E-New, $l \notin \mathit{dom}(S)$. Hence $S[l \mapsto \bot]$ adds a new binding for l in S.

Hence $S[l \mapsto \bot]$ does not update any existing bindings in S.

Hence, for all $l' \in \mathit{dom}(S), S(l') \sqsubseteq (S[l \mapsto \bot])(l')$.

Therefore $S \sqsubseteq_S S[l \mapsto \bot]$, as required.

- Case E-Put:

  Given: $\langle S;\ \texttt{put}\ l\ d_2 \rangle \hookrightarrow \langle S[l \mapsto d_1 \sqcup d_2];\ () \rangle$.

  To show: $S \sqsubseteq_S S[l \mapsto d_1 \sqcup d_2]$.

  By Definition 2.2, we have to show that $\mathit{dom}(S) \subseteq \mathit{dom}(S[l \mapsto d_1 \sqcup d_2])$ and that for all $l' \in \mathit{dom}(S)$, $S(l') \sqsubseteq (S[l \mapsto d_1 \sqcup d_2])(l')$.

  By definition, a store update operation on S can only either update an existing binding in S or extend S with a new binding. Hence $\mathit{dom}(S) \subseteq \mathit{dom}(S[l \mapsto p_2])$.

  From the premises of E-Put, $S(l) = d_1$. Therefore $l \in \mathit{dom}(S)$.

  Hence $S[l \mapsto d_1 \sqcup d_2]$ updates the existing binding for l in S from d_1 to $d_1 \sqcup d_2$.

  By the definition of $\sqcup$, $d_1 \sqsubseteq (d_1 \sqcup d_2)$. $S[l \mapsto d_1 \sqcup d_2]$ does not update any other bindings in S, hence, for all $l' \in \mathit{dom}(S), S(l') \sqsubseteq (S[l \mapsto d_1 \sqcup d_2])(l')$.

  Hence $S \sqsubseteq_S S[l \mapsto d_1 \sqcup d_2]$, as required.

- Case E-Put-Err:

  Given: $\langle S;\ \texttt{put}\ l\ d_2 \rangle \hookrightarrow$ **error**.

  By the definition of **error**, **error** is equal to $\langle \top_S;\ e \rangle$ for all e.

  To show: $S \sqsubseteq_S \top_S$.

  Immediate by the definition of $\sqsubseteq_S$.

- Case E-Get:

  Immediate by the definition of $\sqsubseteq_S$, since S does not change.

□

A.5. Proof of Lemma 2.5

Proof. Consider arbitrary S'' such that S'' is non-conflicting with $\langle S;\ e\rangle \longrightarrow \langle S';\ e'\rangle$ and $S' \sqcup_S S'' \neq \top_S$.

To show: $\langle S \sqcup_S S'';\ e\rangle \longrightarrow \langle S' \sqcup_S S'';\ e'\rangle$.

The proof is by cases on the rule of the reduction semantics by which $\langle S;\ e\rangle$ steps to $\langle S';\ e'\rangle$. Since $\langle S';\ e'\rangle \neq$ **error**, we do not need to consider the E-Put-Err rule.

- Case E-Beta:

  Given: $\langle S;\ (\lambda x.\, e)\ v\rangle \longrightarrow \langle S;\ e[x := v]\rangle$.

  To show: $\langle S \sqcup_S S'';\ (\lambda x.\, e)\ v\rangle \longrightarrow \langle S \sqcup_S S'';\ e[x := v]\rangle$.

  Immediate by E-Beta.

- Case E-New:

  Given: $\langle S;\ \texttt{new}\rangle \longrightarrow \langle S[l \mapsto \bot];\ l\rangle$.

  To show: $\langle S \sqcup_S S'';\ \texttt{new}\rangle \longrightarrow \langle (S[l \mapsto \bot]) \sqcup_S S'';\ l\rangle$.

  By E-New, we have that $\langle S \sqcup_S S'';\ \texttt{new}\rangle \longrightarrow \langle (S \sqcup_S S'')[l' \mapsto \bot];\ l'\rangle$, where $l' \notin \mathit{dom}(S \sqcup_S S'')$.

  By assumption, S'' is non-conflicting with $\langle S;\ \texttt{new}\rangle \longrightarrow \langle S[l \mapsto \bot];\ l\rangle$.

  Therefore $l \notin \mathit{dom}(S'')$.

  From the side condition of E-New, $l \notin \mathit{dom}(S)$.

  Therefore $l \notin \mathit{dom}(S \sqcup_S S'')$.

  Therefore, in $\langle (S \sqcup_S S'')[l' \mapsto \bot];\ l'\rangle$, we can α-rename l' to l,

  resulting in $\langle (S \sqcup_S S'')[l \mapsto \bot];\ l\rangle$.

  Therefore $\langle S \sqcup_S S'';\ \texttt{new}\rangle \longrightarrow \langle (S \sqcup_S S'')[l \mapsto \bot];\ l\rangle$.

Note that:

$$\begin{aligned}(S \sqcup_S S'')[l \mapsto \bot] &= S[l \mapsto \bot] \sqcup_S S''[l \mapsto \bot] \\ &= S \sqcup_S [l \mapsto \bot] \sqcup_S S'' \sqcup_S [l \mapsto \bot] \\ &= S \sqcup_S [l \mapsto \bot] \sqcup_S S'' \\ &= S[l \mapsto \bot] \sqcup_S S''.\end{aligned}$$

Therefore $\langle S \sqcup_S S''; \texttt{new}\rangle \hookrightarrow \langle S[l \mapsto \bot] \sqcup_S S''; l\rangle$, as we were required to show.

- Case E-Put:

  Given: $\langle S; \texttt{put}\ l\ d_2\rangle \hookrightarrow \langle S[l \mapsto d_2]; ()\rangle$.

  To show: $\langle S \sqcup_S S''; \texttt{put}\ l\ d_2\rangle \hookrightarrow \langle S[l \mapsto d_2] \sqcup_S S''; ()\rangle$.

  We will first show that

  $\langle S \sqcup_S S''; \texttt{put}\ l\ d_2\rangle \hookrightarrow \langle (S \sqcup_S S'')[l \mapsto d_2]; ()\rangle$

  and then show why this is sufficient.

  We proceed by cases on l:

  - $l \notin dom(S'')$:

    By assumption, $S[l \mapsto d_2] \sqcup_S S'' \neq \top_S$.

    By Lemma 2.4, $S \sqsubseteq_S S[l \mapsto d_2]$.

    Hence $S \sqcup_S S'' \neq \top_S$.

    Therefore, by Definition 2.3, $(S \sqcup_S S'')(l) = S(l)$.

    From the premises of E-Put, $S(l) = d_1$.

    Hence $(S \sqcup_S S'')(l) = d_1$.

    From the premises of E-Put, $d_2 = d_1 \sqcup d_2$ and $d_2 \neq \top$.

    Therefore, by E-Put, we have: $\langle S \sqcup_S S''; \texttt{put}\ l\ d_2\rangle \hookrightarrow \langle (S \sqcup_S S'')[l \mapsto d_2]; ()\rangle$.

  - $l \in dom(S'')$:

    By assumption, $S[l \mapsto d_2] \sqcup_S S'' \neq \top_S$.

By Lemma 2.4, $S \sqsubseteq_S S[l \mapsto d_2]$.

Hence $S \sqcup_S S'' \neq \top_S$.

Therefore $(S \sqcup_S S'')(l) = S(l) \sqcup S''(l)$.

From the premises of E-Put, $S(l) = d_1$.

Hence $(S \sqcup_S S'')(l) = d_1'$, where $d_1 \sqsubseteq d_1'$.

From the premises of E-Put, $d_2 = d_1 \sqcup d_2$.

Let $d_2' = d_1' \sqcup d_2$.

Hence $d_2 \sqsubseteq d_2'$.

By assumption, $S[l \mapsto d_2] \sqcup_S S'' \neq \top_S$.

Therefore, by Definition 2.3, $d_2 \sqcup_S S''(l) \neq \top$.

Note that:

$$\begin{aligned}
\top &\neq d_2 \sqcup_S S''(l) \\
&= d_1 \sqcup d_2 \sqcup S''(l) \\
&= S(l) \sqcup d_2 \sqcup S''(l) \\
&= S(l) \sqcup S''(l) \sqcup d_2 \\
&= (S \sqcup_S S'')(l) \sqcup d_2 \\
&= d_1' \sqcup d_2 \\
&= d_2'.
\end{aligned}$$

Hence $d_2' \neq \top$.

Hence $(S \sqcup_S S'')(l) = d_1'$ and $d_2' = d_1' \sqcup d_2$ and $d_2' \neq \top$.

Therefore, by E-Put we have: $\langle S \sqcup_S S''; \texttt{put}\ l\ d_2 \rangle \longrightarrow \langle (S \sqcup_S S'')[l \mapsto d_2']; \texttt{()} \rangle$.

LK: If we really wanted to be pedantic here, we'd actually prove that the stores are equal. I'm assuming that if I can show that $(S \sqcup_S S'')[l \mapsto d_2']$ and $(S \sqcup_S S'')[l \mapsto d_2]$ bind l to the same value, then it will be obvious that they're equal.

Note that:

$$\begin{aligned}((S \sqcup_S S'')[l \mapsto d_2'])(l) &= (S \sqcup_S S'')(l) \sqcup ([l \mapsto d_2'])(l) \\ &= d_1' \sqcup d_2' \\ &= d_1' \sqcup d_1' \sqcup d_2 \\ &= d_1' \sqcup d_2\end{aligned}$$

and

$$\begin{aligned}((S \sqcup_S S'')[l \mapsto d_2])(l) &= (S \sqcup_S S'')(l) \sqcup ([l \mapsto d_2])(l) \\ &= d_1' \sqcup d_2 \\ &= d_1' \sqcup d_1 \sqcup d_2 \\ &= d_1' \sqcup d_2 \qquad \text{(since } d_1 \sqsubseteq d_1'\text{)}.\end{aligned}$$

Therefore $(S \sqcup_S S'')[l \mapsto d_2'] = (S \sqcup_S S'')[l \mapsto d_2]$.

Therefore, $\langle S \sqcup_S S''; \texttt{put}\ l\ d_2 \rangle \longhookrightarrow \langle (S \sqcup_S S'')[l \mapsto d_2]; \texttt{()} \rangle$.

Note that:

$$\begin{aligned}(S \sqcup_S S'')[l \mapsto d_2] &= S[l \mapsto d_2] \sqcup_S S''[l \mapsto d_2] \\ &= S \sqcup_S [l \mapsto d_2] \sqcup_S S'' \sqcup_S [l \mapsto d_2] \\ &= S \sqcup_S [l \mapsto d_2] \sqcup_S S'' \\ &= S[l \mapsto d_2] \sqcup_S S''.\end{aligned}$$

Therefore $\langle S \sqcup_S S''; \texttt{put}\ l\ d_2\rangle \longrightarrow \langle S[l \mapsto d_2] \sqcup_S S''; \texttt{()}\rangle$, as we were required to show.

- Case E-Get:

  Given: $\langle S; \texttt{get}\ l\ T\rangle \longrightarrow \langle S; d_2\rangle$.

  To show: $\langle S \sqcup_S S''; \texttt{get}\ l\ T\rangle \longrightarrow \langle S \sqcup_S S''; d_2\rangle$.

  From the premises of E-Get, $S(l) = d_1$ and *incomp*(T) and $d_2 \in T$ and $d_2 \sqsubseteq d_1$.

  By assumption, $S \sqcup_S S'' \neq \top_S$.

  Hence $(S \sqcup_S S'') = d_1'$, where $d_1 \sqsubseteq d_1'$.

  By the transitivity of $\sqsubseteq$, $d_2 \sqsubseteq d_1'$.

  Hence, $S(l) = d_1'$ and *incomp*(T) and $d_2 \in T$ and $d_2 \sqsubseteq d_1'$.

  Therefore, by E-Get,

  $\langle S \sqcup_S S''; \texttt{get}\ l\ T\rangle \longrightarrow \langle S \sqcup_S S''; d_2\rangle$,

  as we were required to show.

□

A.6. Proof of Lemma 2.6

Proof. Consider arbitrary S'' such that S'' is non-conflicting with $\langle S; e\rangle \longrightarrow \langle S'; e'\rangle$ and $S' \sqcup_S S'' = \top_S$.

To show: There exists $i \leq 1$ such that $\langle S \sqcup_S S''; e\rangle \longrightarrow^i$ **error**.

The proof is by cases on the rule of the reduction semantics by which $\langle S; e\rangle$ steps to $\langle S'; e'\rangle$. Since $\langle S'; e'\rangle \neq$ **error**, we do not need to consider the E-Put-Err rule.

- Case E-Beta:

  Given: $\langle S; (\lambda x.\, e)\ v\rangle \longrightarrow \langle S; e[x := v]\rangle$.

  To show: $\langle S \sqcup_S S''; (\lambda x.\, e)\ v\rangle \longrightarrow^i$ **error**, where $i \leq 1$.

  By assumption, $S \sqcup_S S'' = \top_S$.

  Hence, by the definition of **error**, $\langle S \sqcup_S S''; (\lambda x.\, e)\ v\rangle =$ **error**.

Hence $\langle S \sqcup_S S''; (\lambda x.\, e)\, v\rangle \hookrightarrow^i$ **error**, with $i = 0$.

- Case E-New:

  Given: $\langle S; \texttt{new}\rangle \hookrightarrow \langle S[l \mapsto \bot]; l\rangle$.

  To show: $\langle S \sqcup_S S''; \texttt{new}\rangle \hookrightarrow^i$ **error**, where $i \leq 1$.

  By E-New, $\langle S \sqcup_S S''; \texttt{new}\rangle \hookrightarrow \langle (S \sqcup_S S'')[l' \mapsto \bot]; l'\rangle$, where $l' \notin \mathit{dom}(S \sqcup_S S'')$.

  By assumption, S'' is non-conflicting with $\langle S; \texttt{new}\rangle \hookrightarrow \langle S[l \mapsto \bot]; l\rangle$.

  Therefore $l \notin \mathit{dom}(S'')$.

  From the side condition of E-New, $l \notin \mathit{dom}(S)$.

  Therefore $l \notin \mathit{dom}(S \sqcup_S S'')$.

  Therefore, in $\langle (S \sqcup_S S'')[l' \mapsto \bot]; l'\rangle$, we can α-rename l' to l,

  resulting in $\langle (S \sqcup_S S'')[l \mapsto \bot]; l\rangle$.

  Therefore $\langle S \sqcup_S S''; \texttt{new}\rangle \hookrightarrow \langle (S \sqcup_S S'')[l \mapsto \bot]; l\rangle$.

  By assumption, $S[l \mapsto \bot] \sqcup_S S'' = \top_S$.

  Note that:

$$\begin{aligned}
\top_S &= S[l \mapsto \bot] \sqcup_S S'' \\
&= S \sqcup_S [l \mapsto \bot] \sqcup_S S'' \\
&= S \sqcup_S S'' \sqcup_S [l \mapsto \bot] \\
&= (S \sqcup_S S'') \sqcup_S [l \mapsto \bot] \\
&= (S \sqcup_S S'')[l \mapsto \bot].
\end{aligned}$$

  Hence $\langle S \sqcup_S S''; \texttt{new}\rangle \hookrightarrow \langle \top_S; l\rangle$.

  Hence, by the definition of **error**, $\langle S \sqcup_S S''; \texttt{new}\rangle \hookrightarrow$ **error**.

  Hence $\langle S \sqcup_S S''; \texttt{new}\rangle \hookrightarrow^i$ **error**, with $i = 1$.

- Case E-Put:

  Given: $\langle S; \texttt{put}\ l\ d_2\rangle \hookrightarrow \langle S[l \mapsto d_2]; \texttt{()}\rangle$.

To show: $\langle S \sqcup_S S''; \texttt{put}\ l\ d_2\rangle \hookrightarrow^i$ **error**, where $i \leq 1$.

We proceed by cases on $S \sqcup_S S''$:

- $S \sqcup_S S'' = \top_S$:

  In this case, by the definition of **error**, $\langle S \sqcup_S S''; \texttt{put}\ l\ d_2\rangle =$ **error**.

  Hence $\langle S \sqcup_S S''; \texttt{put}\ l\ d_2\rangle \hookrightarrow^i$ **error**, with $i = 0$.

- $S \sqcup_S S'' \neq \top_S$:

  From the premises of E-Put, we have that $S(l) = d_1$.

  Hence $(S \sqcup_S S'')(l) = d_1'$, where $d_1 \sqsubseteq d_1'$.

  We show that $d_1' \sqcup d_2 = \top$, as follows:

  By assumption, $S[l \mapsto d_2] \sqcup_S S'' = \top_S$.

  Hence, by Definition 2.3, there exists some $l' \in \mathit{dom}(S[l \mapsto d_2]) \cap \mathit{dom}(S'')$ such that $(S[l \mapsto d_2])(l') \sqcup S''(l') = \top$.

  Now case on l':

  * $l' \neq l$:

    In this case, $(S[l \mapsto d_2])(l') = S(l')$.

    Since $(S[l \mapsto d_2])(l') \sqcup S''(l') = \top$, we then have that $S(l') \sqcup S''(l') = \top$.

    However, this is a contradiction since $S \sqcup_S S'' \neq \top_S$.

    Hence this case cannot occur.

  * $l' = l$:

    Then $(S[l \mapsto d_2])(l) \sqcup S''(l) = \top$.

Note that:

$$\begin{aligned}\top &= (S[l \mapsto d_2])(l) \sqcup S''(l) \\ &= d_2 \sqcup S''(l) \\ &= d_1 \sqcup d_2 \sqcup S''(l) \\ &= S(l) \sqcup d_2 \sqcup S''(l) \\ &= S(l) \sqcup S''(l) \sqcup d_2 \\ &= (S \sqcup_S S'')(l) \sqcup d_2 \\ &= d_1' \sqcup d_2.\end{aligned}$$

Hence $d_1' \sqcup d_2 = \top$.

Hence, by E-Put-Err, $\langle S \sqcup_S S''; \texttt{put}\ l\ d_2\rangle \hookrightarrow$ **error**.

Hence $\langle S \sqcup_S S''; \texttt{put}\ l\ d_2\rangle \hookrightarrow^i$ **error**, with $i = 1$.

- Case E-Get:

  Given: $\langle S; \texttt{get}\ l\ T\rangle \hookrightarrow \langle S; d_2\rangle$.

  To show: $\langle S \sqcup_S S''; \texttt{get}\ l\ T\rangle \hookrightarrow^i$ **error**, where $i \leq 1$.

  By assumption, $S \sqcup_S S'' = \top_S$.

  Hence, by the definition of **error**, $\langle S \sqcup_S S''; \texttt{get}\ l\ T\rangle =$ **error**.

  Hence $\langle S \sqcup_S S''; \texttt{get}\ l\ T\rangle \hookrightarrow^i$ **error**, with $i = 0$.

□

A.7. Proof of Lemma 2.8

Proof. Suppose $\sigma \longmapsto \sigma_a$ and $\sigma \longmapsto \sigma_b$. We have to show that there exist σ_c, i, j, π such that $\sigma_a \longmapsto^i \sigma_c$ and $\pi(\sigma_b) \longmapsto^j \sigma_c$ and $i \leq 1$ and $j \leq 1$.

By inspection of the operational semantics, it must be the case that σ steps to σ_a by the E-Eval-Ctxt rule. Let $\sigma = \langle S;\ E_a[e_{a_1}]\rangle$ and let $\sigma_a = \langle S_a;\ E_a[e_{a_2}]\rangle$.

Likewise, it must be the case that σ steps to σ_b by the E-Eval-Ctxt rule. Let $\sigma = \langle S;\ E_b[e_{b_1}]\rangle$ and let $\sigma_b = \langle S_b;\ E_b[e_{b_2}]\rangle$.

Note that $\sigma = \langle S;\ E_a[e_{a_1}]\rangle = \langle S;\ E_b[e_{b_1}]\rangle$, and so $E_a[e_{a_1}] = E_b[e_{b_1}]$, but E_a and E_b may differ and e_{a_1} and e_{b_1} may differ.

Since $\langle S;\ E_a[e_{a_1}]\rangle \longmapsto \langle S_a;\ E_a[e_{a_2}]\rangle$ and $\langle S;\ E_b[e_{b_1}]\rangle \longmapsto \langle S_b;\ E_b[e_{b_2}]\rangle$ and $E_a[e_{a_1}] = E_b[e_{b_1}]$, we have from Lemma 2.3 (Locality) that there exist evaluation contexts E'_a and E'_b such that:

- $E'_a[e_{a_1}] = E_b[e_{b_2}]$, and
- $E'_b[e_{b_1}] = E_a[e_{a_2}]$, and
- $E'_a[e_{a_2}] = E'_b[e_{b_2}]$.

In some of the cases that follow, we will choose $\sigma_c =$ **error**. In most cases, however, our approach will be to show that there exist S', i, j, π such that:

- $\langle S_a;\ E_a[e_{a_2}]\rangle \longmapsto^i \langle S';\ E'_a[e_{a_2}]\rangle$, and
- $\pi(\langle S_b;\ E_b[e_{b_2}]\rangle) \longmapsto^j \langle S';\ E'_a[e_{a_2}]\rangle$.

Since $E'_a[e_{a_1}] = E_b[e_{b_2}]$, $E'_b[e_{b_1}] = E_a[e_{a_2}]$, and $E'_a[e_{a_2}] = E'_b[e_{b_2}]$, it suffices to show that:

- $\langle S_a;\ E'_b[e_{b_1}]\rangle \longmapsto^i \langle S';\ E'_b[e_{b_2}]\rangle$, and
- $\pi(\langle S_b;\ E'_a[e_{a_1}]\rangle) \longmapsto^j \langle S';\ E'_a[e_{a_2}]\rangle$.

From the premise of E-Eval-Ctxt, we have that $\langle S;\ e_{a_1}\rangle \hookrightarrow \langle S_a;\ e_{a_2}\rangle$ and $\langle S;\ e_{b_1}\rangle \hookrightarrow \langle S_b;\ e_{b_2}\rangle$. We proceed by case analysis on the rule by which $\langle S;\ e_{a_1}\rangle$ steps to $\langle S_a;\ e_{a_2}\rangle$.

(1) Case E-Beta: We have $S_a = S$.

We proceed by case analysis on the rule by which $\langle S;\ e_{b_1}\rangle$ steps to $\langle S_b;\ e_{b_2}\rangle$:

(a) Case E-Beta: We have $S_b = S$.

Choose $S' = S = S_a = S_b$, $i = 1$, $j = 1$, and $\pi = \text{id}$.

We have to show that:

- $\langle S;\ E'_b[e_{b_1}]\rangle \longmapsto \langle S_a;\ E'_b[e_{b_2}]\rangle$, and
- $\langle S;\ E'_a[e_{a_1}]\rangle \longmapsto \langle S_b;\ E'_a[e_{a_2}]\rangle$,

both of which follow immediately from $\langle S;\ e_{a_1}\rangle \hookrightarrow \langle S_a;\ e_{a_2}\rangle$ and $\langle S;\ e_{b_1}\rangle \hookrightarrow \langle S_b;\ e_{b_2}\rangle$ and E-Eval-Ctxt.

(b) Case E-New: We have $S_b = S[l \mapsto \bot]$.

Choose $S' = S_b$, $i = 1$, $j = 1$, and $\pi = \text{id}$.

We have to show that:

- $\langle S;\ E'_b[e_{b_1}]\rangle \longmapsto \langle S_b;\ E'_b[e_{b_2}]\rangle$, and
- $\langle S_b;\ E'_a[e_{a_1}]\rangle \longmapsto \langle S_b;\ E'_a[e_{a_2}]\rangle$.

The first of these follows immediately from $\langle S;\ e_{b_1}\rangle \hookrightarrow \langle S_b;\ e_{b_2}\rangle$ and E-Eval-Ctxt. For the second, consider that $S_b = S[l \mapsto \bot] = S \sqcup_S [l \mapsto \bot]$. Furthermore, since no locations are allocated during the transition $\langle S;\ e_{a_1}\rangle \hookrightarrow \langle S_a;\ e_{a_2}\rangle$, we know that $[l \mapsto \bot]$ is non-conflicting with it, and we know that $S_a \sqcup_S [l \mapsto \bot] \neq \top_S$ since S_a is just S and $S \sqcup_S [l \mapsto \bot]$ cannot be $\top_S$. Therefore, by Lemma 2.5 (Independence), we have that $\langle S \sqcup_S [l \mapsto \bot]\ ;\ e_{a_1}\rangle \hookrightarrow \langle S_a \sqcup_S [l \mapsto \bot]\ ;\ e_{a_2}\rangle$. Hence $\langle S_b;\ e_{a_1}\rangle \hookrightarrow \langle S_b;\ e_{a_2}\rangle$. By E-Eval-Ctxt, it follows that $\langle S_b;\ E'_a[e_{a_1}]\rangle \longmapsto \langle S_b;\ E'_a[e_{a_2}]\rangle$, as we were required to show.

(c) Case E-Put: We have $S_b = S[l \mapsto d_1 \sqcup d_2]$.

Choose $S' = S_b$, $i = 1$, $j = 1$, and $\pi = \text{id}$.

We have to show that:

- $\langle S;\ E'_b[e_{b_1}]\rangle \longmapsto \langle S_b;\ E'_b[e_{b_2}]\rangle$, and
- $\langle S_b;\ E'_a[e_{a_1}]\rangle \longmapsto \langle S_b;\ E'_a[e_{a_2}]\rangle$.

The first of these follows immediately from $\langle S;\ e_{b_1}\rangle \hookrightarrow \langle S_b;\ e_{b_2}\rangle$ and E-Eval-Ctxt. For the second, consider that $S_b = S[l \mapsto d_1 \sqcup d_2] = S \sqcup_S [l \mapsto d_1 \sqcup d_2]$. Furthermore, since no locations are allocated during the transition $\langle S;\ e_{a_1}\rangle \hookrightarrow \langle S_a;\ e_{a_2}\rangle$, we know that $[l \mapsto d_1 \sqcup d_2]$

is non-conflicting with it, and we know that $S_a \sqcup_S [l \mapsto d_1 \sqcup d_2] \neq \top_S$ since S_a is just S and $S \sqcup_S [l \mapsto d_1 \sqcup d_2]$ cannot be $\top_S$, since we know from the premise of E-Put that $d_1 \sqcup d_2 \neq \top$. Therefore, by Lemma 2.5 (Independence), we have that $\langle S \sqcup_S [l \mapsto d_1 \sqcup d_2]\,;\, e_{a_1}\rangle \hookrightarrow \langle S_a \sqcup_S [l \mapsto d_1 \sqcup d_2]\,;\, e_{a_2}\rangle$. Hence $\langle S_b;\, e_{a_1}\rangle \hookrightarrow \langle S_b;\, e_{a_2}\rangle$. By E-Eval-Ctxt, it follows that $\langle S_b;\, E'_a[e_{a_1}]\rangle \longmapsto \langle S_b;\, E'_a[e_{a_2}]\rangle$, as we were required to show.

(d) Case E-Put-Err:

Here $\langle S_b;\, e_{b_2}\rangle = \textbf{error}$, and so we choose $\sigma_c = \textbf{error}$, $i = 1$, $j = 0$, and $\pi = \text{id}$. We have to show that:

- $\langle S;\, E'_b[e_{b_1}]\rangle \longmapsto \textbf{error}$, and
- $\langle S_b;\, E'_a[e_{a_1}]\rangle = \textbf{error}$.

The second of these is immediately true because since $\langle S_b;\, e_{b_2}\rangle = \textbf{error}$, $S_b = \top_S$, and so $\langle S_b;\, E'_a[e_{a_1}]\rangle$ is equal to **error** as well. For the first, observe that $\langle S;\, e_{b_1}\rangle \hookrightarrow \langle S_b;\, e_{b_2}\rangle$, hence by E-Eval-Ctxt, $\langle S;\, E'_b[e_{b_1}]\rangle \longmapsto \langle S_b;\, E'_b[e_{b_2}]\rangle$. But $S_b = \top_S$, so $\langle S_b;\, E'_b[e_{b_2}]\rangle$ is equal to **error**, and so $\langle S;\, E'_b[e_{b_1}]\rangle \longmapsto \textbf{error}$, as required.

(e) Case E-Get: Similar to case 1a, since $S_b = S$.

(2) Case E-New: We have $S_a = S[l \mapsto \bot]$.

We proceed by case analysis on the rule by which $\langle S;\, e_{b_1}\rangle$ steps to $\langle S_b;\, e_{b_2}\rangle$:

(a) Case E-Beta: By symmetry with case 1b.

(b) Case E-New: We have $S_b = S[l' \mapsto \bot]$.

Now consider whether $l = l'$:

- If $l \neq l'$:

  Choose $S' = S[l' \mapsto \bot][l \mapsto \bot]$, $i = 1$, $j = 1$, and $\pi = \text{id}$.

  We have to show that:

  - $\langle S_a;\, E'_b[e_{b_1}]\rangle \longmapsto \langle S[l' \mapsto \bot][l \mapsto \bot];\, E'_b[e_{b_2}]\rangle$, and
  - $\langle S_b;\, E'_a[e_{a_1}]\rangle \longmapsto \langle S[l' \mapsto \bot][l \mapsto \bot];\, E'_a[e_{a_2}]\rangle$.

For the first of these, consider that $S_a = S[l \mapsto \bot] = S \sqcup_S [l \mapsto \bot]$, and that $S[l' \mapsto \bot][l \mapsto \bot] = S[l' \mapsto \bot] \sqcup_S [l \mapsto \bot]$. Furthermore, since the only location allocated during the transition $\langle S;\ e_{b_1}\rangle \longhookrightarrow \langle S_b;\ e_{b_2}\rangle$ is l', we know that $[l \mapsto \bot]$ is non-conflicting with it (since $l \neq l'$ in this case). We also know that $S[l' \mapsto \bot] \sqcup_S [l \mapsto \bot] \neq \top_S$, since $S \neq \top_S$ and new bindings of $l \mapsto \bot$ and $l' \mapsto \bot$ cannot cause it to become $\top_S$. Therefore, by Lemma 2.5 (Independence), we have that $\langle S \sqcup_S [l \mapsto \bot]\ ;\ e_{b_1}\rangle \longhookrightarrow \langle S_b \sqcup_S [l \mapsto \bot]\ ;\ e_{b_2}\rangle$. Hence $\langle S[l \mapsto \bot];\ e_{b_1}\rangle \longhookrightarrow \langle S_b[l \mapsto \bot];\ e_{b_2}\rangle$. By E-Eval-Ctxt it follows that $\langle S[l \mapsto \bot];\ E'_b[e_{b_1}]\rangle \longhookrightarrow \langle S_b[l \mapsto \bot];\ E'_b[e_{b_2}]\rangle$, which, since $S_b = S[l' \mapsto \bot]$, is what we were required to show. The argument for the second is symmetrical.

- If $l = l'$:

  In this case, observe that we do *not* want the expression in the final configuration to be $E'_a[e_{a_2}]$ (nor its equivalent, $E'_b[e_{b_2}]$). The reason for this is that $E'_a[e_{a_2}]$ contains both occurrences of l. Rather, we want both configurations to step to a configuration in which exactly one occurrence of l has been renamed to a fresh location l''.

  Let l'' be a location such that $l'' \notin S$ and $l'' \neq l$ (and hence $l'' \neq l'$, as well). Then choose $S' = S[l'' \mapsto \bot][l \mapsto \bot]$, $i = 1$, $j = 1$, and $\pi = \{(l, l'')\}$.

  Either $\langle S[l'' \mapsto \bot][l \mapsto \bot];\ E'_a[\pi(e_{a_2})]\rangle$ or $\langle S[l'' \mapsto \bot][l \mapsto \bot];\ E'_b[\pi(e_{b_2})]\rangle$ would work as a final configuration; we choose $\langle S[l'' \mapsto \bot][l \mapsto \bot];\ E'_b[\pi(e_{b_2})]\rangle$.

  We have to show that:

  - $\langle S_a;\ E'_b[e_{b_1}]\rangle \longmapsto \langle S[l'' \mapsto \bot][l \mapsto \bot];\ E'_b[\pi(e_{b_2})]\rangle$, and
  - $\pi(\langle S_b;\ E'_a[e_{a_1}]\rangle) \longmapsto \langle S[l'' \mapsto \bot][l \mapsto \bot];\ E'_b[\pi(e_{b_2})]\rangle$.

  For the first of these, since $\langle S;\ e_{b_1}\rangle \longhookrightarrow \langle S_b;\ e_{b_2}\rangle$, we have by Lemma 2.1 (Permutability) that $\pi(\langle S;\ e_{b_1}\rangle) \longhookrightarrow \pi(\langle S_b;\ e_{b_2}\rangle)$. Since $\pi = \{(l, l'')\}$, but $l \notin S$ (from the side condition on E-New), we have that $\pi(\langle S;\ e_{b_1}\rangle) = \langle S;\ e_{b_1}\rangle$. Since $\langle S_b;\ e_{b_2}\rangle =$

$\langle S[l' \mapsto \bot];\, l' \rangle$, and $l = l'$, we have that $\pi(\langle S_b;\, e_{b_2} \rangle) = \langle S[l'' \mapsto \bot];\, \pi(e_{b_2}) \rangle$. Hence $\langle S;\, e_{b_1} \rangle \longhookrightarrow \langle S[l'' \mapsto \bot];\, \pi(e_{b_2}) \rangle$.

Since the only location allocated during the transition $\langle S;\, e_{b_1} \rangle \longhookrightarrow \langle S[l'' \mapsto \bot];\, \pi(e_{b_2}) \rangle$ is l'', we know that $[l \mapsto \bot]$ is non-conflicting with it. We also know that $S[l'' \mapsto \bot] \sqcup_S [l \mapsto \bot] \neq \top_S$, since $S \neq \top_S$ and new bindings of $l'' \mapsto \bot$ and $l \mapsto \bot$ cannot cause it to become $\top_S$. Therefore, by Lemma 2.5 (Independence), we have that $\langle S \sqcup_S [l \mapsto \bot]\,;\, e_{b_1} \rangle \longhookrightarrow \langle S[l'' \mapsto \bot] \sqcup_S [l \mapsto \bot]\,;\, \pi(e_{b_2}) \rangle$. Hence $\langle S[l \mapsto \bot];\, e_{b_1} \rangle \longhookrightarrow \langle S[l'' \mapsto \bot][l \mapsto \bot];\, \pi(e_{b_2}) \rangle$. By E-Eval-Ctxt it follows that $\langle S[l \mapsto \bot];\, E'_b[e_{b_1}] \rangle \longhookrightarrow \langle S[l'' \mapsto \bot][l \mapsto \bot];\, E'_b[\pi(e_{b_2})] \rangle$, which, since $S[l \mapsto \bot] = S_a$, is what we were required to show.

For the second, observe that since $S_b = S[l \mapsto \bot]$, we have that $\pi(S_b) = S[l'' \mapsto \bot]$. Also, since l does not occur in e_{a_1}, we have that $\pi(E'_a[e_{a_1}]) = (\pi(E'_a))[e_{a_1}]$. Hence we have to show that

$\langle S[l'' \mapsto \bot];\, (\pi(E'_a))[e_{a_1}] \rangle \longmapsto \langle S[l'' \mapsto \bot][l \mapsto \bot];\, E'_b[\pi(e_{b_2})] \rangle$.

Since the only location allocated during the transition $\langle S;\, e_{a_1} \rangle \longhookrightarrow \langle S_a;\, e_{a_2} \rangle$ is l, we know that $[l'' \mapsto \bot]$ is non-conflicting with it. We also know that $S_a \sqcup_S [l'' \mapsto \bot] \neq \top_S$, since $S_a = S[l \mapsto \bot]$ and $S \neq \top_S$ and new bindings of $l'' \mapsto \bot$ and $l \mapsto \bot$ cannot cause it to become $\top_S$. Therefore, by Lemma 2.5 (Independence), we have that $\langle S \sqcup_S [l'' \mapsto \bot]\,;\, e_{a_1} \rangle \longhookrightarrow \langle S_a \sqcup_S [l'' \mapsto \bot]\,;\, e_{a_2} \rangle$. Hence $\langle S[l'' \mapsto \bot];\, e_{a_1} \rangle \longhookrightarrow \langle S[l'' \mapsto \bot][l \mapsto \bot];\, e_{a_2} \rangle$. By E-Eval-Ctxt it follows that $\langle S[l'' \mapsto \bot];\, (\pi(E'_a))[e_{a_1}] \rangle \longmapsto \langle S[l'' \mapsto \bot][l \mapsto \bot];\, (\pi(E'_a))[e_{a_2}] \rangle$, which completes the case since $E'_b[\pi(e_{b_2})] = (\pi(E'_a))[e_{a_2}]$.

LK: This is really sketchy – I should really explain why $E'_b[\pi(e_{b_2})] = (\pi(E'_a))[e_{a_2}]$.

(c) Case E-Put: We have $S_b = S[l' \mapsto d_1 \sqcup d_2]$.

We have to show that:

- $\langle S_a;\, E'_b[e_{b_1}] \rangle \longmapsto \langle S_b[l \mapsto \bot];\, E'_b[e_{b_2}] \rangle$, and

- $\langle S_b;\ E'_a[e_{a_1}]\rangle \longmapsto \langle S_b[l \mapsto \bot];\ E'_a[e_{a_2}]\rangle$.

For the first of these, consider that $S_a = S[l \mapsto \bot] = S \sqcup_S [l \mapsto \bot]$, and that $S_b[l \mapsto \bot] = S_b \sqcup_S [l \mapsto \bot]$. Furthermore, since no locations are allocated during the transition $\langle S;\ e_{b_1}\rangle \hookrightarrow \langle S_b;\ e_{b_2}\rangle$, we know that $[l \mapsto \bot]$ is non-conflicting with it. We also know that $S_b \sqcup_S [l \mapsto \bot] \neq \top_S$, since $S_b = S[l' \mapsto d_1 \sqcup d_2]$ and we know from the premise of E-Put that $d_1 \sqcup d_2 \neq \top$. Therefore, by Lemma 2.5 (Independence), we have that $\langle S \sqcup_S [l \mapsto \bot]\ ;\ e_{b_1}\rangle \hookrightarrow \langle S_b \sqcup_S [l \mapsto \bot]\ ;\ e_{b_2}\rangle$. Hence $\langle S[l \mapsto \bot];\ e_{b_1}\rangle \hookrightarrow \langle S_b[l \mapsto \bot];\ e_{b_2}\rangle$. By E-Eval-Ctxt, it follows that $\langle S[l \mapsto \bot];\ E'_b[e_{b_1}]\rangle \longmapsto \langle S_b[l \mapsto \bot];\ E'_b[e_{b_2}]\rangle$, which, since $S_a = S[l \mapsto \bot]$, is what we were required to show.

For the second, consider that $S_b = S \sqcup_S [l' \mapsto d_1 \sqcup d_2]$ and $S_b[l \mapsto \bot] = S[l \mapsto \bot] \sqcup_S [l' \mapsto d_1 \sqcup d_2] = S_a \sqcup_S [l' \mapsto d_1 \sqcup d_2]$. Furthermore, since the only location allocated during the transition $\langle S;\ e_{a_1}\rangle \hookrightarrow \langle S_a;\ e_{a_2}\rangle$ is l, we know that $[l' \mapsto d_1 \sqcup d_2]$ is non-conflicting with it. (We know that $l \neq l'$ because we have from the premise of E-Put that $l' \in \mathit{dom}(S)$, but we have from the side condition of E-New that $l \notin \mathit{dom}(S)$.) We also know that $S[l \mapsto \bot] \sqcup_S [l' \mapsto d_1 \sqcup d_2] \neq \top_S$, since we know from the premise of E-Put that $d_1 \sqcup d_2 \neq \top$. Therefore, by Lemma 2.5 (Independence), we have that $\langle S \sqcup_S [l' \mapsto d_1 \sqcup d_2]\ ;\ e_{a_1}\rangle \hookrightarrow \langle S_a \sqcup_S [l' \mapsto d_1 \sqcup d_2]\ ;\ e_{a_2}\rangle$. Hence $\langle S_b;\ e_{a_1}\rangle \hookrightarrow \langle S_b[l \mapsto \bot];\ e_{a_2}\rangle$. By E-Eval-Ctxt, it follows that $\langle S_b;\ E'_a[e_{a_1}]\rangle \longmapsto \langle S_b[l \mapsto \bot];\ E'_a[e_{a_2}]\rangle$, as we were required to show.

(d) Case E-Put-Err:

Here $\langle S_b;\ e_{b_2}\rangle = \textbf{error}$, and so we choose $\sigma_c = \textbf{error}$, $i = 1$, $j = 0$, and $\pi = \mathrm{id}$. We have to show that:

- $\langle S_a;\ E'_b[e_{b_1}]\rangle \longmapsto \textbf{error}$, and
- $\langle S_b;\ E'_a[e_{a_1}]\rangle = \textbf{error}$.

The second of these is immediately true because since $\langle S_b;\ e_{b_2}\rangle = \textbf{error}$, $S_b = \top_S$, and so $\langle S_b;\ E'_a[e_{a_1}]\rangle$ is equal to **error** as well. For the first, observe that since $\langle S;\ e_{a_1}\rangle \hookrightarrow \langle S_a;\ e_{a_2}\rangle$, we have by Lemma 2.4 (Monotonicity) that $S \sqsubseteq_S S_a$. Therefore, since $\langle S;\ e_{b_1}\rangle \hookrightarrow$

error, we have by Lemma 2.7 (Error Preservation) that $\langle S_a;\ e_{b_1}\rangle \longrightarrow$ **error**. Since **error** is equal to $\langle \top_S;\ e\rangle$ for all expressions e, $\langle S_a;\ e_{b_1}\rangle \longrightarrow \langle \top_S;\ e\rangle$ for all e. Therefore, by E-Eval-Ctxt, $\langle S_a;\ E'_b[e_{b_1}]\rangle \longmapsto \langle \top_S;\ E'_b[e]\rangle$ for all e. Since $\langle \top_S;\ E'_b[e]\rangle$ is equal to **error**, we have that $\langle S_a;\ E'_b[e_{b_1}]\rangle \longmapsto$ **error**, as we were required to show.

(e) Case E-Get: Similar to case 2a, since $S_b = S$.

(3) Case E-Put: We have $S_a = S[l \mapsto d_1 \sqcup d_2]$.

We proceed by case analysis on the rule by which $\langle S;\ e_{b_1}\rangle$ steps to $\langle S_b;\ e_{b_2}\rangle$:

(a) Case E-Beta: By symmetry with case 1c.

(b) Case E-New: By symmetry with case 2c.

(c) Case E-Put: We have $S_b = S[l' \mapsto d'_1 \sqcup d'_2]$, where $d'_1 = S(l')$.

Consider whether $S_b \sqcup_S [l \mapsto d_1 \sqcup d_2] = \top_S$:

- $S_b \sqcup_S [l \mapsto d_1 \sqcup d_2] \neq \top_S$:

  Choose $S' = S_a \sqcup_S S_b$, $i = 1$, $j = 1$, and $\pi = \mathrm{id}$.

  We have to show that:

  – $\langle S_a;\ E'_b[e_{b_1}]\rangle \longmapsto \langle S_a \sqcup_S S_b;\ E'_b[e_{b_2}]\rangle$, and

  – $\langle S_b;\ E'_a[e_{a_1}]\rangle \longmapsto \langle S_a \sqcup_S S_b;\ E'_a[e_{a_2}]\rangle$.

  For the first of these, since no locations are allocated during the transition $\langle S;\ e_{b_1}\rangle \longrightarrow \langle S_b;\ e_{b_2}\rangle$, we know that $[l \mapsto d_1 \sqcup d_2]$ is non-conflicting with it, and in this subcase, we know that $S_b \sqcup_S [l \mapsto d_1 \sqcup d_2] \neq \top_S$. Therefore, by Lemma 2.5 (Independence), we have that $\langle S \sqcup_S [l \mapsto d_1 \sqcup d_2];\ e_{b_1}\rangle \longrightarrow \langle S_b \sqcup_S [l \mapsto d_1 \sqcup d_2];\ e_{b_2}\rangle$. By E-Eval-Ctxt, it follows that $\langle S \sqcup_S [l \mapsto d_1 \sqcup d_2];\ E'_b[e_{b_1}]\rangle \longmapsto \langle S_b \sqcup_S [l \mapsto d_1 \sqcup d_2];\ E'_b[e_{b_2}]\rangle$. Since $S \sqcup_S [l \mapsto d_1 \sqcup d_2] = S[l \mapsto d_1 \sqcup d_2] = S_a$, we have that $\langle S_a;\ E'_b[e_{b_1}]\rangle \longmapsto \langle S_b \sqcup_S [l \mapsto d_1 \sqcup d_2];\ E'_b[e_{b_2}]\rangle$. Furthermore, since $\langle S;\ e_{b_1}\rangle \longrightarrow \langle S_b;\ e_{b_2}\rangle$, by Lemma 2.4 (Monotonicity), we have that $S \sqsubseteq_S S_b$, so $S_b \sqcup_S [l \mapsto d_1 \sqcup d_2] = S_b \sqcup_S S \sqcup_S [l \mapsto d_1 \sqcup d_2] = S_b \sqcup_S S_a = S_a \sqcup_S S_b$. So we have that $\langle S_a;\ E'_b[e_{b_1}]\rangle \longmapsto \langle S_a \sqcup_S S_b;\ E'_b[e_{b_2}]\rangle$, as we were required to show.

The argument for the second is symmetrical, with $[l' \mapsto d_1' \sqcup d_2']$ being the transition that is non-conflicting with $\langle S;\ e_{a_1}\rangle \hookrightarrow \langle S_a;\ e_{a_2}\rangle$.

- $S_b \sqcup_S [l \mapsto d_1 \sqcup d_2] = \top_S$:

  Here we choose $\sigma_c =$ **error** and $\pi =$ id. We have to show that there exist $i \leq 1$ and $j \leq 1$ such that:

  - $\langle S_a;\ E_b'[e_{b_1}]\rangle \longmapsto^i$ **error**, and
  - $\langle S_b;\ E_a'[e_{a_1}]\rangle \longmapsto^j$ **error**.

  For the first of these, since no locations are allocated during the transition $\langle S;\ e_{b_1}\rangle \hookrightarrow \langle S_b;\ e_{b_2}\rangle$, we know that $[l \mapsto d_1 \sqcup d_2]$ is non-conflicting with it, and in this subcase, we know that $S_b \sqcup_S [l \mapsto d_1 \sqcup d_2] = \top_S$. Therefore, by Lemma 2.6 (Clash), we have that $\langle S \sqcup_S [l \mapsto d_1 \sqcup d_2]\ ;\ e_{b_1}\rangle \hookrightarrow^{i'}$ **error**, where $i' \leq 1$. Since **error** is equal to $\langle \top_S;\ e\rangle$ for all expressions e, $\langle S \sqcup_S [l \mapsto d_1 \sqcup d_2]\ ;\ e_{b_1}\rangle \hookrightarrow^{i'} \langle \top_S;\ e\rangle$ for all e.

  Now consider whether $i' = 1$ or $i' = 0$:

  - If $i' = 1$, by E-Eval-Ctxt, it follows that $\langle S \sqcup_S [l \mapsto d_1 \sqcup d_2]\ ;\ E_b'[e_{b_1}]\rangle \longmapsto \langle \top_S;\ E_b'[e]\rangle$ for all e. Since $\langle \top_S;\ E_b'[e]\rangle$ is equal to **error**, and since $S \sqcup_S [l \mapsto d_1 \sqcup d_2] = S[l \mapsto d_1 \sqcup d_2] = S_a$, we choose $i = 1$ and we have that $\langle S_a;\ E_b'[e_{b_1}]\rangle \longmapsto$ **error**, as required.
  - If $i' = 0$, we have that $\langle S \sqcup_S [l \mapsto d_1 \sqcup d_2]\ ;\ e_{b_1}\rangle =$ **error**. Hence $S \sqcup_S [l \mapsto d_1 \sqcup d_2] = \top_S$. So, we choose $i = 0$, and since $S_a = S[l \mapsto d_1 \sqcup d_2] = S \sqcup_S [l \mapsto d_1 \sqcup d_2] = \top_S$, we have that $\langle S_a;\ E_b'[e_{b_1}]\rangle =$ **error**, as desired.

  The argument for the second is symmetrical, with $[l' \mapsto d_1' \sqcup d_2']$ being the transition that is non-conflicting with $\langle S;\ e_{a_1}\rangle \hookrightarrow \langle S_a;\ e_{a_2}\rangle$.

(d) Case E-Put-Err:

Here $\langle S_b;\ e_{b_2}\rangle =$ **error**, and so we choose $\sigma_c =$ **error**, $i = 1$, $j = 0$, and $\pi =$ id. We have to show that:

- $\langle S_a;\ E_b'[e_{b_1}]\rangle \longmapsto$ **error**, and

- $\langle S_b;\ E'_a[e_{a_1}]\rangle = \textbf{error}$.

The second of these is immediately true because since $\langle S_b;\ e_{b_2}\rangle = \textbf{error}$, $S_b = \top_S$, and so $\langle S_b;\ E'_a[e_{a_1}]\rangle$ is equal to **error** as well. For the first, observe that since $\langle S;\ e_{a_1}\rangle \hookrightarrow \langle S_a;\ e_{a_2}\rangle$, we have by Lemma 2.4 (Monotonicity) that $S \sqsubseteq_S S_a$. Therefore, since $\langle S;\ e_{b_1}\rangle \hookrightarrow$ **error**, we have by Lemma 2.7 (Error Preservation) that $\langle S_a;\ e_{b_1}\rangle \hookrightarrow$ **error**. Since **error** is equal to $\langle \top_S;\ e\rangle$ for all expressions e, $\langle S_a;\ e_{b_1}\rangle \hookrightarrow \langle \top_S;\ e\rangle$ for all e. Therefore, by E-Eval-Ctxt, $\langle S_a;\ E'_b[e_{b_1}]\rangle \longmapsto \langle \top_S;\ E'_b[e]\rangle$ for all e. Since $\langle \top_S;\ E'_b[e]\rangle$ is equal to **error**, we have that $\langle S_a;\ E'_b[e_{b_1}]\rangle \longmapsto$ **error**, as we were required to show.

(e) Case E-Get: Similar to case 3a, since $S_b = S$.

(4) Case E-Put-Err: We have $\langle S_a;\ e_{a_2}\rangle = \textbf{error}$.

We proceed by case analysis on the rule by which $\langle S;\ e_{b_1}\rangle$ steps to $\langle S_b;\ e_{b_2}\rangle$:

(a) Case E-Beta: By symmetry with case 1d.

(b) Case E-New: By symmetry with case 2d.

(c) Case E-Put: By symmetry with case 3d.

(d) Case E-Put-Err:

Here $\langle S_b;\ e_{b_2}\rangle = \textbf{error}$, and so we choose $\sigma_c = \textbf{error}$, $i = 0$, $j = 0$, and $\pi = \text{id}$. We have to show that:

- $\langle S_a;\ E'_b[e_{b_1}]\rangle = \textbf{error}$, and
- $\langle S_b;\ E'_a[e_{a_1}]\rangle = \textbf{error}$.

Since $\langle S_a;\ e_{a_2}\rangle = \textbf{error}$, $S_a = \top_S$, and since $\langle S_b;\ e_{b_2}\rangle = \textbf{error}$, $S_b = \top_S$, so both of the above follow immediately.

(e) Case E-Get: Similar to case 4a, since $S_b = S$.

(5) Case E-Get:

We proceed by case analysis on the rule by which $\langle S;\ e_{b_1}\rangle$ steps to $\langle S_b;\ e_{b_2}\rangle$:

(a) Case E-Beta: By symmetry with case 1e.

(b) Case E-New: By symmetry with case 2e.

(c) Case E-Put: By symmetry with case 3e.

(d) Case E-Put-Err: By symmetry with case 4e.

(e) Case E-Get: Similar to case 5a, since $S_b = S$.

□

A.8. Proof of Lemma 2.9

Proof. We proceed by induction on m. In the base case of $m = 1$, the result is immediate from Lemma 2.8.

For the induction step, suppose $\sigma \longmapsto^m \sigma'' \longmapsto \sigma'''$ and suppose the lemma holds for m.

We show that it holds for $m+1$, as follows.

We are required to show that there exist σ_c, i, j, π such that $\sigma' \longmapsto^i \sigma_c$ and $\pi(\sigma''') \longmapsto^j \sigma_c$ and $i \leq m+1$ and $j \leq 1$.

From the induction hypothesis, we have that there exist σ'_c, i', j', π' such that $\sigma' \longmapsto^{i'} \sigma'_c$ and $\pi'(\sigma'') \longmapsto^{j'} \sigma'_c$ and $i' \leq m$ and $j' \leq 1$.

We proceed by cases on j':

- If $j' = 0$, then $\pi'(\sigma'') = \sigma'_c$.

  Since $\sigma'' \longmapsto \sigma'''$, we have that $\pi'(\sigma'') \longmapsto \pi'(\sigma''')$ by Lemma 2.1 (Permutability).

  We can then choose $\sigma_c = \pi'(\sigma''')$ and $i = i' + 1$ and $j = 0$ and $\pi = \pi'$. The key is that $\sigma' \longmapsto^{i'} \sigma'_c = \pi'(\sigma'') \longmapsto \pi'(\sigma''')$ for a total of $i'+1$ steps.

- If $j' = 1$:

  First, since $\pi'(\sigma'') \longmapsto^{j'} \sigma'_c$, then by Lemma 2.1 (Permutability) we have that $\sigma'' \longmapsto^{j'} \pi'^{-1}(\sigma'_c)$.

  Then, by $\sigma'' \longmapsto^{j'} \pi'^{-1}(\sigma'_c)$ and $\sigma'' \longmapsto \sigma'''$ and Lemma 2.8 (Strong Local Confluence), we have that there exist σ''_c and i'' and j'' and π'' such that $\pi'^{-1}(\sigma'_c) \longmapsto^{i''} \sigma''_c$ and $\pi''(\sigma''') \longmapsto^{j''} \sigma''_c$ and $i'' \leq 1$ and $j'' \leq 1$.

Since $\pi'^{-1}(\sigma'_c) \longmapsto^{i''} \sigma''_c$, by Lemma 2.1 (Permutability) we have that $\sigma'_c \longmapsto^{i''} \pi'(\sigma''_c)$.

So we also have $\sigma' \longmapsto^{i'} \sigma'_c \longmapsto^{i''} \pi'(\sigma''_c)$.

Since $\pi''(\sigma''') \longmapsto^{j''} \sigma''_c$, by Lemma 2.1 (Permutability) we have that $\pi'(\pi''(\sigma''')) \longmapsto^{j''} \pi'(\sigma''_c)$.

In summary, we pick $\sigma_c = \pi'(\sigma''_c)$ and $i = i' + i''$ and $j = j''$ and $\pi = \pi'' \circ \pi'$, which is sufficient because $i = i' + i'' \leq m + 1$ and $j = j'' \leq 1$.

□

A.9. Proof of Lemma 2.10

Proof. We proceed by induction on n. In the base case of $n = 1$, the result is immediate from Lemma 2.9.

For the induction step, suppose $\sigma \longmapsto^{n} \sigma' \longmapsto \sigma'''$ and suppose the lemma holds for n.

We show that it holds for $n + 1$, as follows.

We are required to show that there exist σ_c, i, j, π such that $\sigma''' \longmapsto^{i} \sigma_c$ and $\pi(\sigma'') \longmapsto^{j} \sigma_c$ and $i \leq m$ and $j \leq n + 1$.

From the induction hypothesis, we have that there exist σ'_c, i', j', π' such that $\sigma' \longmapsto^{i'} \sigma'_c$ and $\pi'(\sigma'') \longmapsto^{j'} \sigma'_c$ and $i' \leq m$ and $j' \leq n$.

We proceed by cases on i':

- If $i' = 0$, then $\sigma' = \sigma'_c$. We can then choose $\sigma_c = \sigma'''$ and $i = 0$ and $j = j' + 1$ and $\pi = \pi'$. Since $\pi'(\sigma'') \longmapsto^{j'} \sigma'_c \longmapsto \sigma'''$, and $j' + 1 \leq n + 1$ since $j' \leq n$, the case is satisfied.
- If $i' \geq 1$:

  From $\sigma' \longmapsto \sigma'''$ and $\sigma' \longmapsto^{i'} \sigma'_c$ and Lemma 2.9, we have that there exist σ''_c and i'' and j'' and π'' such that $\sigma''' \longmapsto^{i''} \sigma''_c$ and $\pi''(\sigma'_c) \longmapsto^{j''} \sigma''_c$ and $i'' \leq i'$ and $j'' \leq 1$.

  Since $\pi'(\sigma'') \longmapsto^{j'} \sigma'_c$, by Lemma 2.1 (Permutability) we have that $\pi''(\pi'(\sigma'')) \longmapsto^{j'} \pi''(\sigma'_c)$.

  So we also have $\pi''(\pi'(\sigma'')) \longmapsto^{j'} \pi''(\sigma'_c) \longmapsto^{j''} \sigma''_c$.

In summary, we pick $\sigma_c = \sigma_c''$ and $i = i''$ and $j = j' + j''$ and $\pi = \pi' \circ \pi''$, which is sufficient because $i = i'' \leq i' \leq m$ and $j = j' + j'' \leq n + 1$.

□

A.10. Proof of Lemma 3.2

Proof.

(1) $\sqsubseteq_p$ is a partial order over D_p.

To show this, we need to show that $\sqsubseteq_p$ is reflexive, transitive, and antisymmetric.

(a) $\sqsubseteq_p$ is reflexive.

Suppose $v \in D_p$. Then, by Lemma 3.1, either $v = (d, \text{false})$ with $d \in D$, or $v = (x, \text{true})$ with $x \in X$, where $X = D - \{\top\}$.

- Suppose $v = (d, \text{false})$:

  By the reflexivity of $\sqsubseteq$, we know $d \sqsubseteq d$.

  By the definition of $\sqsubseteq_p$, we know $(d, \text{false}) \sqsubseteq_p (d, \text{false})$.

- Suppose $v = (x, \text{true})$:

  By the reflexivity of equality, $x = x$.

  By the definition of $\sqsubseteq_p$, we know $(x, \text{true}) \sqsubseteq_p (x, \text{true})$.

(b) $\sqsubseteq_p$ is transitive.

Suppose $v_1 \sqsubseteq_p v_2$ and $v_2 \sqsubseteq_p v_3$. We want to show that $v_1 \sqsubseteq_p v_3$. We proceed by case analysis on v_1, v_2, and v_3.

- Case $v_1 = (d_1, \text{false})$ and $v_2 = (d_2, \text{false})$ and $v_3 = (d_3, \text{false})$:

  By inversion on $\sqsubseteq_p$, it follows that $d_1 \sqsubseteq d_2$.

  By inversion on $\sqsubseteq_p$, it follows that $d_2 \sqsubseteq d_3$.

  By the transitivity of $\sqsubseteq$, we know $d_1 \sqsubseteq d_3$.

By the definition of $\sqsubseteq_p$, it follows that $(d_1, \text{false}) \sqsubseteq_p (d_3, \text{false})$.

Hence $v_1 \sqsubseteq_p v_3$.

- Case $v_1 = (d_1, \text{false})$ and $v_2 = (d_2, \text{false})$ and $v_3 = (x_3, \text{true})$:

  By inversion on $\sqsubseteq_p$, it follows that $d_1 \sqsubseteq d_2$.

  By inversion on $\sqsubseteq_p$, it follows that $d_2 \sqsubseteq x_3$.

  By the transitivity of $\sqsubseteq$, we know $d_1 \sqsubseteq x_3$.

  By the definition of $\sqsubseteq_p$, it follows that $(d_1, \text{false}) \sqsubseteq_p (x_3, \text{true})$.

  Hence $v_1 \sqsubseteq_p v_3$.

- Case $v_1 = (d_1, \text{false})$ and $v_2 = (x_2, \text{true})$ and $v_3 = (d_3, \text{false})$:

  By inversion on $\sqsubseteq_p$, it follows that $d_1 \sqsubseteq x_2$.

  By inversion on $\sqsubseteq_p$, it follows that $d_3 = \top$.

  Since $\top$ is the maximal element of D, we know $d_1 \sqsubseteq \top \equiv d_3$.

  By the definition of $\sqsubseteq_p$, it follows that $(d_1, \text{false}) \sqsubseteq_p (d_3, \text{false})$.

  Hence $v_1 \sqsubseteq_p v_3$.

- Case $v_1 = (d_1, \text{false})$ and $v_2 = (x_2, \text{true})$ and $v_3 = (x_3, \text{true})$:

  By inversion on $\sqsubseteq_p$, it follows that $d_1 \sqsubseteq x_2$.

  By inversion on $\sqsubseteq_p$, it follows that $x_2 = x_3$.

  Hence $d_1 \sqsubseteq x_3$.

  By the definition of $\sqsubseteq_p$, it follows that $(d_1, \text{false}) \sqsubseteq_p (x_3, \text{true})$.

  Hence $v_1 \sqsubseteq_p v_3$.

- Case $v_1 = (x_1, \text{true})$ and $v_2 = (d_2, \text{false})$ and $v_3 = (d_3, \text{false})$:

  By inversion on $\sqsubseteq_p$, it follows that $d_2 = \top$.

  By inversion on $\sqsubseteq_p$, it follows that $d_2 \sqsubseteq d_3$.

  Since $\top$ is maximal, it follows that $d_3 = \top$.

  By the definition of $\sqsubseteq_p$, it follows that $(x_1, \text{true}) \sqsubseteq_p (d_3, \text{false})$.

  Hence $v_1 \sqsubseteq_p v_3$.

- Case $v_1 = (x_1, \text{true})$ and $v_2 = (d_2, \text{false})$ and $v_3 = (x_3, \text{true})$:

  By inversion on $\sqsubseteq_p$, it follows that $d_2 = \top$.

  By inversion on $\sqsubseteq_p$, it follows that $d_2 \sqsubseteq x_3$.

  Since $\top$ is maximal, it follows that $x_3 = \top$.

  But since $x_3 \in X \subseteq D/\{\top\}$, we know $x_3 \neq \top$.

  This is a contradiction.

  Hence $v_1 \sqsubseteq_p v_3$.

- Case $v_1 = (x_1, \text{true})$ and $v_2 = (x_2, \text{true})$ and $v_3 = (d_3, \text{false})$:

  By inversion on $\sqsubseteq_p$, it follows that $x_1 = x_2$.

  By inversion on $\sqsubseteq_p$, it follows that $d_3 = \top$.

  By the definition of $\sqsubseteq_p$, it follows that $(x_1, \text{true}) \sqsubseteq_p (d_3, \text{false})$.

  Hence $v_1 \sqsubseteq_p v_3$.

- Case $v_1 = (x_1, \text{true})$ and $v_2 = (x_2, \text{true})$ and $v_3 = (x_3, \text{true})$:

  By inversion on $\sqsubseteq_p$, it follows that $x_1 = x_2$.

  By inversion on $\sqsubseteq_p$, it follows that $x_2 = x_3$.

  By transitivity of $=$, $x_1 = x_3$.

  By the definition of $\sqsubseteq_p$, it follows that $(x_1, \text{true}) \sqsubseteq_p (x_3, \text{true})$.

  Hence $v_1 \sqsubseteq_p v_3$.

(c) $\sqsubseteq_p$ is antisymmetric.

Suppose $v_1 \sqsubseteq_p v_2$ and $v_2 \sqsubseteq_p v_1$. Now, we proceed by cases on v_1 and v_2.

- Case $v_1 = (d_1, \text{false})$ and $v_2 = (d_2, \text{false})$:

  By inversion on $v_1 \sqsubseteq_p v_2$, we know that $d_1 \sqsubseteq d_2$.

  By inversion on $v_2 \sqsubseteq_p v_1$, we know that $d_2 \sqsubseteq d_1$.

  By the antisymmetry of $\leq$, we know $d_1 = d_2$.

  Hence $v_1 = v_2$.

- Case $v_1 = (d_1, \text{false})$ and $v_2 = (x_2, \text{true})$:

By inversion on $v_1 \sqsubseteq_p v_2$, we know that $d_1 \sqsubseteq x_2$.

By inversion on $v_2 \sqsubseteq_p v_1$, we know that $d_1 = \top$.

Since $\top$ is maximal in D, we know $x_2 = \top$.

But since $x_2 \in X \subseteq D/\{\top\}$, we know $x_2 \neq \top$.

This is a contradiction.

Hence $v_1 = v_2$.

- Case $v_1 = (x_1, \text{true})$ and $v_2 = (d_2, \text{false})$:

  Similar to the previous case.

- Case $v_1 = (x_1, \text{true})$ and $v_2 = (x_2, \text{true})$:

  By inversion on $v_1 \sqsubseteq_p v_2$, we know that $x_1 = x_2$.

  Hence $v_1 = v_2$.

(2) Every nonempty finite subset of D_p has a least upper bound.

To show this, it is sufficient to show that every two elements of D_p have a least upper bound, since a binary least upper bound operation can be repeatedly applied to compute the least upper bound of any finite set. We will show that every two elements of D_p have a least upper bound by showing that the $\sqcup_p$ operation defined by Definition 3.2 computes their least upper bound.

It suffices to show the following two properties:

(a) For all $v_1, v_2, v \in D_p$, if $v_1 \sqsubseteq_p v$ and $v_2 \sqsubseteq_p v$, then $(v_1 \sqcup_p v_2) \sqsubseteq_p v$.

(b) For all $v_1, v_2 \in D_p$, $v_1 \sqsubseteq_p (v_1 \sqcup_p v_2)$ and $v_2 \sqsubseteq_p (v_1 \sqcup_p v_2)$.

(a) For all $v_1, v_2, v \in D_p$, if $v_1 \sqsubseteq_p v$ and $v_2 \sqsubseteq_p v$, then $v_1 \sqcup_p v_2 \sqsubseteq_p v$.

Assume $v_1, v_2, v \in D_p$, and $v_1 \sqsubseteq_p v$ and $v_2 \sqsubseteq_p v$. Now we do a case analysis on v_1 and v_2.

- Case $v_1 = (d_1, \text{false})$ and $v_2 = (d_2, \text{false})$.

  Now case on v:

  - Case $v = (d, \text{false})$:

    By the definition of $\sqcup_p$, $(d_1, \text{false}) \sqcup_p (d_2, \text{false}) = (d_1 \sqcup d_2, \text{false})$.

    By inversion on $(d_1, \text{false}) \sqsubseteq_p (d, \text{false})$, $d_1 \sqsubseteq l$.

By inversion on $(d_2, \text{false}) \sqsubseteq_p (d, \text{false})$, $d_2 \sqsubseteq l$.

Hence l is an upper bound for d_1 and d_2.

Hence $d_1 \sqcup d_2 \sqsubseteq l$.

Hence $(d_1 \sqcup d_2, \text{false}) \sqsubseteq_p (d, \text{false})$.

Hence $v_1 \sqcup_p v_2 \sqsubseteq_p v$.

- Case $v = (x, \text{true})$:

  By the definition of $\sqcup_p$, $(d_1, \text{false}) \sqcup_p (d_2, \text{false}) = (d_1 \sqcup d_2, \text{false})$.

  By inversion on $(d_1, \text{false}) \sqsubseteq_p (x, \text{true})$, $d_1 \sqsubseteq x$.

  By inversion on $(d_2, \text{false}) \sqsubseteq_p (x, \text{true})$, $d_2 \sqsubseteq x$.

  Hence x is an upper bound for d_1 and d_2.

  Hence $d_1 \sqcup d_2 \sqsubseteq x$.

  Hence $(d_1 \sqcup d_2, \text{false}) \sqsubseteq_p (x, \text{true})$.

  Hence $v_1 \sqcup_p v_2 \sqsubseteq_p v$.

- Case $v_1 = (x_1, \text{true})$ and $v_2 = (x_2, \text{true})$:

  Now case on v:

  - Case $v = (d, \text{false})$:

    By inversion on $(x_1, \text{true}) \sqsubseteq_p (d, \text{false})$, we know $l = \top$.

    By inversion on $(x_2, \text{true}) \sqsubseteq_p (d, \text{false})$, we know $l = \top$.

    Now consider whether $x_1 = x_2$ or not. If it does, then by the definition of $\sqcup_p$, $(x_1, \text{true}) \sqcup_p (x_2, \text{true}) = (x_1, \text{true})$.

    By definition of $\sqsubseteq_p$, we have $(x_1, \text{true}) \sqsubseteq_p (\top, \text{false})$. So $v_1 \sqcup_p v_2 \sqsubseteq_p v$.

    If it does not, then $v_1 \sqcup_p v_2 = (\top, \text{false})$.

    By the definition of $\sqsubseteq_p$, we have $(\top, \text{false}) \sqsubseteq_p (\top, \text{false})$. So $v_1 \sqcup_p v_2 \sqsubseteq_p v$.

  - Case $v = (x, \text{true})$:

    By inversion on $(x_1, \text{true}) \sqsubseteq_p (x, \text{true})$, we know $x = x_1$.

    By inversion on $(x_2, \text{true}) \sqsubseteq_p (x, \text{true})$, we know $x = x_2$.

Hence $x_1 = x_2$.

By the definition of $\sqcup_p$, $(x_1, \text{true}) \sqcup_p (x_2, \text{true}) = (x_1, \text{true})$.

Hence $v_1 \sqcup_p v_2 \sqsubseteq_p v$.

- Case $v_1 = (x_1, \text{true})$ and $v_2 = (d_2, \text{false})$:

  Now case on v:

  - Case $v = (d, \text{false})$:

    Now consider whether $d_2 \sqsubseteq x_1$.

    If it is, then $(x_1, \text{true}) \sqcup_p (d_2, \text{false}) = (x_1, \text{true}) = v_1$.

    Hence $v_1 \sqcup_p v_2 \sqsubseteq_p v$.

    Otherwise, $(x_1, \text{true}) \sqcup_p (d_2, \text{false}) = (\top, \text{false})$.

    By inversion on $(x_1, \text{true}) \sqsubseteq_p (d, \text{false})$, we know $l = \top$.

    By reflexivity, $(\top, \text{false}) \sqsubseteq_p (\top, \text{false})$.

    Hence $v_1 \sqcup_p v_2 \sqsubseteq_p v$.

  - Case $v = (x, \text{true})$:

    By inversion on $(x_1, \text{true}) \sqsubseteq_p (x, \text{true})$, we know that $x_1 = x$.

    By inversion on $(d_2, \text{false}) \sqsubseteq_p (x, \text{true})$, we know that $d_2 \sqsubseteq x$.

    By transitivity, $d_2 \sqsubseteq x_1$.

    By the definition of $\sqcup_p$, it follows that $(x_1, \text{true}) \sqcup_p (d_2, \text{false}) = (x_1, \text{true})$.

    By defininition of $\sqsubseteq_p$, $(x_1, \text{true}) \sqsubseteq_p (x_1, \text{true})$.

    Hence $v_1 \sqcup_p v_2 \sqsubseteq_p v$.

- Case $v_1 = (d_1, \text{false})$ and $v_2 = (x_2, \text{true})$:

  Symmetric with the previous case.

(b) For all $v_1, v_2 \in D_p$, $v_1 \sqsubseteq_p v_1 \sqcup_p v_2$ and $v_2 \sqsubseteq_p v_1 \sqcup_p v_2$.

Assume $v_1, v_2 \in D_p$, and proceed by case analysis.

- Case $v_1 = (d_1, \text{false})$ and $v_2 = (d_2, \text{false})$:

Since $\sqcup$ is a join operator, we know $d_1 \sqsubseteq d_1 \sqcup d_2$.

By the definition of $\sqsubseteq_p$, $(d_1, \text{false}) \sqsubseteq (d_1 \sqcup d_2, \text{false})$.

By the definition of $\sqcup_p$, $v_1 \sqcup_p v_2 = (d_1 \sqcup d_2, \text{false})$.

Hence $v_1 \sqsubseteq_p v_1 \sqcup_p v_2$.

Since $\sqcup$ is a join operator, we know $d_1 \sqsubseteq d_1 \sqcup d_2$.

By the definition of $\sqsubseteq_p$, $(d_2, \text{false}) \sqsubseteq (d_1 \sqcup d_2, \text{false})$.

By the definition of $\sqcup_p$, $v_1 \sqcup_p v_2 = (d_1 \sqcup d_2, \text{false})$.

Hence $v_2 \sqsubseteq_p v_1 \sqcup_p v_2$.

Therefore $v_1 \sqsubseteq_p v_1 \sqcup v_2$ and $v_2 \sqsubseteq_p v_1 \sqcup v_2$.

- Case $v_1 = (d_1, \text{false})$ and $v_2 = (x_2, \text{true})$:

  Consider whether $d_1 \sqsubseteq x_2$.

  - Case $d_1 \sqsubseteq x_2$:

    By the definition of $\sqcup_p$, we know $(d_1, \text{false}) \sqcup_p (x_2, \text{true}) = (x_2, \text{true})$.

    By the definition of $\sqcup_p$, we know $(d_1, \text{false}) \sqsubseteq_p (x_2, \text{true})$.

    Hence $v_1 \sqsubseteq_p v_1 \sqcup_p v_2$.

    By reflexivity, $(x_2, \text{true}) \sqsubseteq_p (x_2, \text{true})$.

    Hence $v_2 \sqsubseteq_p v_1 \sqcup_p v_2$.

    Therefore $v_1 \sqsubseteq_p v_1 \sqcup v_2$ and $v_2 \sqsubseteq_p v_1 \sqcup v_2$.

  - Case $d_1 \not\sqsubseteq x_2$:

    By the definition of $\sqcup_p$, we know $(d_1, \text{false}) \sqcup_p (x_2, \text{true}) = (\top, \text{false})$.

    Since $d_1 \sqsubseteq \top$, by the definition of $\sqsubseteq_p$ we know $(d_1, \text{false}) \sqsubseteq (\top, \text{false})$.

    Hence $v_1 \sqsubseteq_p v_1 \sqcup_p v_2$.

    By the definition of $\sqsubseteq_p$, we know $(x_2, \text{true}) \sqsubseteq (\top, \text{false})$.

    Hence $v_2 \sqsubseteq_p v_1 \sqcup_p v_2$.

    Therefore $v_1 \sqsubseteq_p v_1 \sqcup v_2$ and $v_2 \sqsubseteq_p v_1 \sqcup v_2$.

- Case $v_1 = (x_1, \text{true})$ and $v_2 = (d_2, \text{false})$:

  Symmetric with the previous case.

- Case $v_1 = (x_1, \text{true})$ and $v_2 = (x_2, \text{true})$:

  Consider whether x_1 equals x_2.

  - Case $x_1 = x_2$:

    By the definition $\sqcup_p$, $(x_1, \text{true}) \sqcup_p (x_2, \text{true}) = (x_1, \text{true})$.

    By reflexivity, $(x_1, \text{true}) \sqsubseteq_p (x_1, \text{true})$.

    Hence $v_1 \sqsubseteq_p v_1 \sqcup_p v_2$.

    By reflexivity, $(x_2, \text{true}) \sqsubseteq_p (x_1, \text{true})$.

    Hence $v_2 \sqsubseteq_p v_1 \sqcup_p v_2$.

    Therefore $v_1 \sqsubseteq_p v_1 \sqcup v_2$ and $v_2 \sqsubseteq_p v_1 \sqcup v_2$.

  - Case $x_1 \neq x_2$:

    By the definition $\sqcup_p$, $(x_1, \text{true}) \sqcup_p (x_2, \text{true}) = (\top, \text{false})$.

    By the definition of $\sqsubseteq_p$, $(x_1, \text{true}) \sqsubseteq_p (\top, \text{false})$.

    Hence $v_1 \sqsubseteq_p v_1 \sqcup_p v_2$.

    By the definition of $\sqsubseteq_p$, $(x_2, \text{true}) \sqsubseteq_p (\top, \text{false})$.

    Hence $v_2 \sqsubseteq_p v_1 \sqcup_p v_2$.

    Therefore $v_1 \sqsubseteq_p v_1 \sqcup v_2$ and $v_2 \sqsubseteq_p v_1 \sqcup v_2$.

(3) $\bot_p$ is the least element of D_p.

$\bot_p$ is defined to be $(\bot, \text{false})$. In order to be the least element of D_p, it must be less than or equal to every element of D_p. By Lemma 3.1, the elements of D_p partition into (d, false) for all $d \in D$, and (x, true) for all $x \in X$, where $X = D - \{\top\}$.

We consider both cases:

- (d, false) for all $d \in D$:

By the definition of $\sqsubseteq_p$, $(\bot, \text{false}) \sqsubseteq_p (d, \text{false})$ iff $\bot \sqsubseteq d$.

Since $\bot$ is the least element of D, $\bot \sqsubseteq d$.

Therefore $\bot_p = (\bot, \text{false}) \sqsubseteq_p (d, \text{false})$.

- (x, true) for all $x \in X$:

  By the definition of $\sqsubseteq_p$, $(\bot, \text{false}) \sqsubseteq_p (x, \text{true})$ iff $\bot \sqsubseteq x$.

  Since $\bot$ is the least element of D, $\bot \sqsubseteq x$.

  Therefore $\bot_p = (\bot, \text{false}) \sqsubseteq_p (x, \text{true})$.

Therefore $\bot_p$ is less than or equal to all elements of D_p.

(4) $\top_p$ is the greatest element of D_p.

$\top_p$ is defined to be $(\top, \text{false})$. In order to be the greatest element of D_p, every element of D_p must be less than or equal to it. By Lemma 3.1, the elements of D_p partition into (d, false) for all $d \in D$, and (x, true) for all $x \in X$, where $X = D - \{\top\}$.

We consider both cases:

- (d, false) for all $d \in D$:

  By the definition of $\sqsubseteq_p$, $(d, \text{false}) \sqsubseteq_p (\top, \text{false})$ iff $d \sqsubseteq \top$.

  Since $\top$ is the greatest element of D, $d \sqsubseteq \top$.

  Therefore $(d, \text{false}) \sqsubseteq_p (\top, \text{false}) = \top_p$.

- (x, true) for all $x \in X$:

  By the definition of $\sqsubseteq_p$, $(x, \text{true}) \sqsubseteq_p (\top, \text{false})$ iff $\top \sqsubseteq \top$.

  Therefore $(x, \text{true}) \sqsubseteq_p (\top, \text{false}) = \top_p$.

Therefore all elements of D_p are less than or equal to $\top_p$.

□

A.11. Proof of Lemma 3.3

Proof. TODO: Prove this. □

A.12. Proof of Lemma 3.4

Proof. TODO: Prove this. □

A.13. Proof of Lemma 3.5

Proof. TODO: Needs to be updated.

- Case E-Eval-Ctxt:

  Given: $\langle S;\ E[e]\rangle \longrightarrow \langle S';\ E[e']\rangle$.

  To show: $S \sqsubseteq_S S'$.

  From the premise of E-Eval-Ctxt, $\langle S;\ e\rangle \longrightarrow \langle S';\ e'\rangle$.

  Hence by IH, $S \sqsubseteq_S S'$, as we were required to show.

- Case E-Beta:

  Immediate by the definition of $\sqsubseteq_S$, since S does not change.

- Case E-New:

  Given: $\langle S;\ \texttt{new}\rangle \longrightarrow \langle S[l \mapsto (\bot, \text{false})];\ l\rangle$.

  To show: $S \sqsubseteq_S S[l \mapsto (\bot, \text{false})]$.

  By Definition 3.4, we have to show that $\mathit{dom}(S) \subseteq \mathit{dom}(S[l \mapsto (\bot, \text{false})])$ and that for all $l' \in \mathit{dom}(S)$,

  $S(l') \sqsubseteq_p (S[l \mapsto (\bot, \text{false})])(l')$.

  By the definition of store update, $S[l \mapsto (d_1, \text{true})]$ can only either update an existing binding in S or extend S with a new binding.

  Hence $\mathit{dom}(S) \subseteq \mathit{dom}(S[l \mapsto (\bot, \text{false})])$.

  From the side condition of E-New, $l \notin \mathit{dom}(S)$.

  Hence $S[l \mapsto (\bot, \text{false})]$ adds a new binding for l in S.

  Hence $S[l \mapsto (d_1, \text{true})]$ does not update any existing bindings in S.

  Hence, for all $l' \in \mathit{dom}(S), S(l') \sqsubseteq_p (S[l \mapsto (d_1, \text{true})])(l')$.

Therefore $S \sqsubseteq_S S[l \mapsto (\bot, \text{false})]$, as required.

- Case E-Put:

  Given: $\langle S;\ \texttt{put}\ l\ d_2 \rangle \longrightarrow \langle S[l \mapsto p_2];\ () \rangle$.

  To show: $S \sqsubseteq_S S[l \mapsto p_2]$.

  By Definition 3.4, we have to show that $\mathit{dom}(S) \subseteq \mathit{dom}(S[l \mapsto p_2])$ and that for all $l' \in \mathit{dom}(S)$, $S(l') \sqsubseteq_p (S[l \mapsto p_2])(l')$.

  By the definition of store update, $S[l \mapsto p_2]$ can only either update an existing binding in S or extend S with a new binding.

  Hence $\mathit{dom}(S) \subseteq \mathit{dom}(S[l \mapsto p_2])$.

  From the premises of E-Put, $S(l) = p_1$. Therefore $l \in \mathit{dom}(S)$.

  Hence $S[l \mapsto p_2]$ updates the existing binding for l in S from p_1 to p_2.

  From the premises of E-Put, $p_2 = p_1 \sqcup_p (d_2, \text{false})$.

  Hence, by the definition of $\sqcup_p$, $p_1 \sqsubseteq_p p_2$.

  $S[l \mapsto p_2]$ does not update any other bindings in S, hence, for all $l' \in \mathit{dom}(S)$, $S(l') \sqsubseteq_p (S[l \mapsto p_2])(l')$.

  Hence $S \sqsubseteq_S S[l \mapsto p_2]$, as required.

- Case E-Put-Err:

  Given: $\langle S;\ \texttt{put}\ l\ d_2 \rangle \longrightarrow \textbf{error}$.

  By the definition of **error**, $\textbf{error} = \langle \top_S;\ e \rangle$ for any e.

  To show: $S \sqsubseteq_S \top_S$.

  Immediate by the definition of $\sqsubseteq_S$.

- Case E-Get:

  Immediate by the definition of $\sqsubseteq_S$, since S does not change.

- Case E-Freeze-Init:

  Immediate by the definition of $\sqsubseteq_S$, since S does not change.

- Case E-Spawn-Handler:

Immediate by the definition of $\sqsubseteq_S$, since S does not change.

- Case E-Freeze-Final:

  Given: $\langle S;\ \texttt{freeze}\ l\ \texttt{after}\ Q\ \texttt{with}\ v, \{v \ldots\}, H\rangle \longrightarrow \langle S[l \mapsto (d_1, \text{true})];\ d_1\rangle$.

  To show: $S \sqsubseteq_S S[l \mapsto (d_1, \text{true})]$.

  By Definition 3.4, we have to show that $\mathit{dom}(S) \subseteq \mathit{dom}(S[l \mapsto (d_1, \text{true})])$ and that for all $l' \in \mathit{dom}(S)$,

  $S(l') \sqsubseteq_p (S[l \mapsto (d_1, \text{true})])(l')$.

  LK: We could spell this out in even more excruciating detail, but I think it's obvious enough.

  By the definition of store update, $S[l \mapsto (d_1, \text{true})]$ can only either update an existing binding in S or extend S with a new binding.

  Hence $\mathit{dom}(S) \subseteq \mathit{dom}(S[l \mapsto (d_1, \text{true})])$.

  From the premises of E-Freeze-Final, $S(l) = (d_1, \mathit{frz}_1)$. Therefore $l \in \mathit{dom}(S)$.

  Hence $S[l \mapsto (d_1, \text{true})]$ updates the existing binding for l in S from (d_1, frz_1) to (d_1, true).

  By the definition of $\sqsubseteq_p$, $(d_1, \mathit{frz}_1) \sqsubseteq_p (d_1, \text{true})$.

  $S[l \mapsto (d_1, \text{true})]$ does not update any other bindings in S, hence, for all $l' \in \mathit{dom}(S)$,

  $S(l') \sqsubseteq_p (S[l \mapsto (d_1, \text{true})])(l')$.

  Hence $S \sqsubseteq_S S[l \mapsto (d_1, \text{true})]$, as required.

- Case E-Freeze-Simple:

  Given: $\langle S;\ \texttt{freeze}\ l\rangle \longrightarrow \langle S[l \mapsto (d_1, \text{true})];\ d_1\rangle$.

  To show: $S \sqsubseteq_S S[l \mapsto (d_1, \text{true})]$.

  Similar to the previous case.

□

A.14. Proof of Lemma 3.6

Proof. TODO: Needs to be updated.

Consider arbitrary S'' such that S'' is non-conflicting with $\langle S;\ e\rangle \longrightarrow \langle S';\ e'\rangle$ and $S' \sqcup_S S'' =_{frz} S$ and $S' \sqcup_S S'' \neq \top_S$.

To show: $\langle S \sqcup_S S'';\ e\rangle \longrightarrow \langle S' \sqcup_S S'';\ e'\rangle$.

The proof is by induction on the derivation of $\langle S;\ e\rangle \longrightarrow \langle S';\ e'\rangle$, by cases on the last rule in the derivation. In every case we may assume that $\langle S';\ e'\rangle \neq$ **error**. Since $\langle S';\ e'\rangle \neq$ **error**, we do not need to consider the E-Put-Err rule.

The assumption that $S' \sqcup_S S'' =_{frz} S$ is only needed in the E-Freeze-Final and E-Freeze-Simple cases.

- Case E-Eval-Ctxt:

  Given: $\langle S;\ E[e]\rangle \longrightarrow \langle S';\ E[e']\rangle$.

  To show: $\langle S \sqcup_S S'';\ E[e]\rangle \longrightarrow \langle S' \sqcup_S S'';\ E[e']\rangle$.

  From the premise of E-Eval-Ctxt, we have that $\langle S;\ e\rangle \longrightarrow \langle S';\ e'\rangle$.

  Therefore, by IH, we have that $\langle S \sqcup_S S'';\ e\rangle \longrightarrow \langle S' \sqcup_S S'';\ e'\rangle$.

  Therefore, by E-Eval-Ctxt, we have that $\langle S \sqcup_S S'';\ E[e]\rangle \longrightarrow \langle S' \sqcup_S S'';\ E[e']\rangle$, as we were required to show.

- Case E-Beta:

  Given: $\langle S;\ (\lambda x.\, e)\ v\rangle \longrightarrow \langle S;\ e[x := v]\rangle$.

  To show: $\langle S \sqcup_S S'';\ (\lambda x.\, e)\ v\rangle \longrightarrow \langle S \sqcup_S S'';\ e[x := v]\rangle$.

  Immediate by E-Beta.

- Case E-New:

  Given: $\langle S;\ \texttt{new}\rangle \longrightarrow \langle S[l \mapsto (\bot, \text{false})];\ l\rangle$.

  To show: $\langle S \sqcup_S S'';\ \texttt{new}\rangle \longrightarrow \langle (S[l \mapsto (\bot, \text{false})]) \sqcup_S S'';\ l\rangle$.

  By E-New, we have that $\langle S \sqcup_S S'';\ \texttt{new}\rangle \longrightarrow \langle (S \sqcup_S S'')[l' \mapsto (\bot, \text{false})];\ l'\rangle$, where $l' \notin \mathit{dom}(S \sqcup_S S'')$.

  By assumption, S'' is non-conflicting with $\langle S;\ \texttt{new}\rangle \longrightarrow \langle S[l \mapsto (\bot, \text{false})];\ l\rangle$.

  Therefore $l \notin \mathit{dom}(S'')$.

From the side condition of E-New, $l \notin \mathit{dom}(S)$.

Therefore $l \notin \mathit{dom}(S \sqcup_S S'')$.

Therefore, in $\langle (S \sqcup_S S'')[l' \mapsto (\bot, \text{false})];\ l' \rangle$, we can α-rename l' to l,

resulting in $\langle (S \sqcup_S S'')[l \mapsto (\bot, \text{false})];\ l \rangle$.

Therefore $\langle S \sqcup_S S''; \texttt{new} \rangle \longrightarrow \langle (S \sqcup_S S'')[l \mapsto (\bot, \text{false})];\ l \rangle$.

Note that:

$$\begin{aligned}(S \sqcup_S S'')[l \mapsto (\bot, \text{false})] &= S[l \mapsto (\bot, \text{false})] \sqcup_S S''[l \mapsto (\bot, \text{false})] \\ &= S \sqcup_S [l \mapsto (\bot, \text{false})] \sqcup_S S'' \sqcup_S [l \mapsto (\bot, \text{false})] \\ &= S \sqcup_S [l \mapsto (\bot, \text{false})] \sqcup_S S'' \\ &= S[l \mapsto (\bot, \text{false})] \sqcup_S S''.\end{aligned}$$

Therefore $\langle S \sqcup_S S''; \texttt{new} \rangle \longrightarrow \langle S[l \mapsto (\bot, \text{false})] \sqcup_S S'';\ l \rangle$, as we were required to show.

- Case E-Put:

  LK: The assumption that $S[l \mapsto p_2] \sqcup_S S'' \neq \top_S$ rules out the put-after-freeze possibility.

  Given: $\langle S;\ \texttt{put}\ l\ d_2 \rangle \longrightarrow \langle S[l \mapsto p_2];\ () \rangle$.

  To show: $\langle S \sqcup_S S'';\ \texttt{put}\ l\ d_2 \rangle \longrightarrow \langle S[l \mapsto p_2] \sqcup_S S'';\ () \rangle$.

  We will first show that

  $\langle S \sqcup_S S'';\ \texttt{put}\ l\ d_2 \rangle \longrightarrow \langle (S \sqcup_S S'')[l \mapsto p_2];\ () \rangle$

  and then show why this is sufficient.

  We proceed by cases on l:

  - $l \notin \mathit{dom}(S'')$:

    By assumption, $S[l \mapsto p_2] \sqcup_S S'' \neq \top_S$.

    By Lemma 3.5, $S \sqsubseteq_S S[l \mapsto p_2]$.

    Hence $S \sqcup_S S'' \neq \top_S$.

    Therefore, by Definition 3.5, $(S \sqcup_S S'')(l) = S(l)$.

From the premises of E-Put, $S(l) = p_1$.

Hence $(S \sqcup_S S'')(l) = p_1$.

From the premises of E-Put, $p_2 = p_1 \sqcup_p (d_2, \text{false})$ and $p_2 \neq \top_p$.

Therefore, by E-Put, we have: $\langle S \sqcup_S S''; \texttt{put}\ l\ d_2\rangle \hookrightarrow \langle (S \sqcup_S S'')[l \mapsto p_2];\ ()\rangle$.

– $l \in \mathit{dom}(S'')$:

By assumption, $S[l \mapsto p_2] \sqcup_S S'' \neq \top_S$.

By Lemma 3.5, $S \sqsubseteq_S S[l \mapsto p_2]$.

Hence $S \sqcup_S S'' \neq \top_S$.

Therefore $(S \sqcup_S S'')(l) = S(l) \sqcup_p S''(l)$.

From the premises of E-Put, $S(l) = p_1$.

Hence $(S \sqcup_S S'')(l) = p_1'$, where $p_1 \sqsubseteq_p p_1'$.

From the premises of E-Put, $p_2 = p_1 \sqcup_p (d_2, \text{false})$.

Let $p_2' = p_1' \sqcup_p (d_2, \text{false})$.

Hence $p_2 \sqsubseteq_p p_2'$.

By assumption, $S[l \mapsto p_2] \sqcup_S S'' \neq \top_S$.

Therefore, by Definition 3.5, $p_2 \sqcup_S S''(l) \neq \top_p$.

Note that:

$$\begin{aligned}
\top_p &\neq p_2 \sqcup_S S''(l) \\
&= p_1 \sqcup_p (d_2, \mathsf{false}) \sqcup_p S''(l) \\
&= S(l) \sqcup_p (d_2, \mathsf{false}) \sqcup_p S''(l) \\
&= S(l) \sqcup_p S''(l) \sqcup_p (d_2, \mathsf{false}) \\
&= (S \sqcup_S S'')(l) \sqcup_p (d_2, \mathsf{false}) \\
&= p_1' \sqcup_p (d_2, \mathsf{false}) \\
&= p_2'.
\end{aligned}$$

Hence $p_2' \neq \top_p$.

Hence $(S \sqcup_S S'')(l) = p_1'$ and $p_2' = p_1' \sqcup_p (d_2, \mathsf{false})$ and $p_2' \neq \top_p$.

Therefore, by E-Put we have: $\langle S \sqcup_S S''; \texttt{put}\ l\ d_2\rangle \longrightarrow \langle (S \sqcup_S S'')[l \mapsto p_2']; \texttt{()}\rangle$.

LK: If we really wanted to be pedantic here, we'd actually prove that the stores are equal. I'm assuming that if I can show that $(S \sqcup_S S'')[l \mapsto p_2']$ and $(S \sqcup_S S'')[l \mapsto p_2]$ bind l to the same value, then it will be obvious that they're equal.

Note that:

$$\begin{aligned}
((S \sqcup_S S'')[l \mapsto p_2'])(l) &= (S \sqcup_S S'')(l) \sqcup_p ([l \mapsto p_2'])(l) \\
&= p_1' \sqcup_p p_2' \\
&= p_1' \sqcup_p p_1' \sqcup_p (d_2, \mathsf{false}) \\
&= p_1' \sqcup_p (d_2, \mathsf{false})
\end{aligned}$$

and

$$\begin{aligned}((S \sqcup_S S'')[l \mapsto p_2])(l) &= (S \sqcup_S S'')(l) \sqcup_p ([l \mapsto p_2])(l) \\ &= p_1' \sqcup_p p_2 \\ &= p_1' \sqcup_p p_1 \sqcup_p (d_2, \text{false}) \\ &= p_1' \sqcup_p (d_2, \text{false}) \qquad \text{(since } p_1 \sqsubseteq_p p_1'\text{)}.\end{aligned}$$

Therefore $(S \sqcup_S S'')[l \mapsto p_2'] = (S \sqcup_S S'')[l \mapsto p_2]$.

Therefore, $\langle S \sqcup_S S''; \texttt{put}\ l\ d_2\rangle \longhookrightarrow \langle (S \sqcup_S S'')[l \mapsto p_2];\ \texttt{()}\rangle$.

Note that:

$$\begin{aligned}(S \sqcup_S S'')[l \mapsto p_2] &= S[l \mapsto p_2] \sqcup_S S''[l \mapsto p_2] \\ &= S \sqcup_S [l \mapsto p_2] \sqcup_S S'' \sqcup_S [l \mapsto p_2] \\ &= S \sqcup_S [l \mapsto p_2] \sqcup_S S'' \\ &= S[l \mapsto p_2] \sqcup_S S''.\end{aligned}$$

Therefore $\langle S \sqcup_S S''; \texttt{put}\ l\ d_2\rangle \longhookrightarrow \langle S[l \mapsto p_2] \sqcup_S S'';\ \texttt{()}\rangle$, as we were required to show.

- Case E-Get:

  Given: $\langle S; \texttt{get}\ l\ P\rangle \longhookrightarrow \langle S;\ p_2\rangle$.

  To show: $\langle S \sqcup_S S''; \texttt{get}\ l\ P\rangle \longhookrightarrow \langle S \sqcup_S S'';\ p_2\rangle$.

  From the premises of E-Get, $S(l) = p_1$ and *incomp*(P) and $p_2 \in P$ and $p_2 \sqsubseteq_p p_1$.

  By assumption, $S \sqcup_S S'' \neq \top_S$.

  Hence $(S \sqcup_S S'') = p_1'$, where $p_1 \sqsubseteq_p p_1'$.

  By the transitivity of $\sqsubseteq_p$, $p_2 \sqsubseteq_p p_1'$.

  Hence, $S(l) = p_1'$ and *incomp*(P) and $p_2 \in P$ and $p_2 \sqsubseteq_p p_1'$.

  Therefore, by E-Get,

$\langle S \sqcup_S S''; \texttt{get}\ l\ P\rangle \hookrightarrow \langle S \sqcup_S S''; p_2\rangle$,

as we were required to show.

- Case E-Freeze-Init:

  Given: $\langle S; \texttt{freeze}\ l\ \texttt{after}\ Q\ \texttt{with}\ \lambda x.\, e\rangle \hookrightarrow$

  $\langle S; \texttt{freeze}\ l\ \texttt{after}\ Q\ \texttt{with}\ \lambda x.\, e, \{\}, \{\}\rangle$.

  To show: $\langle S \sqcup_S S''; \texttt{freeze}\ l\ \texttt{after}\ Q\ \texttt{with}\ \lambda x.\, e\rangle \hookrightarrow$

  $\langle S \sqcup_S S''; \texttt{freeze}\ l\ \texttt{after}\ Q\ \texttt{with}\ \lambda x.\, e, \{\}, \{\}\rangle$.

  Immediate by E-Freeze-Init.

- Case E-Spawn-Handler:

  Given:

  $\langle S; \texttt{freeze}\ l\ \texttt{after}\ Q\ \texttt{with}\ \lambda x.\, e_0, \{e, \dots\}, H\rangle \hookrightarrow$

  $\langle S; \texttt{freeze}\ l\ \texttt{after}\ Q\ \texttt{with}\ \lambda x.\, e_0, \{e_0[x := d_2], e, \dots\}, \{d_2\} \cup H\rangle$.

  To show:

  $\langle S \sqcup_S S''; \texttt{freeze}\ l\ \texttt{after}\ Q\ \texttt{with}\ \lambda x.\, e_0, \{e, \dots\}, H\rangle \hookrightarrow$

  $\langle S \sqcup_S S''; \texttt{freeze}\ l\ \texttt{after}\ Q\ \texttt{with}\ \lambda x.\, e_0, \{e_0[x := d_2], e, \dots\}, \{d_2\} \cup H\rangle$.

  From the premises of E-Spawn-Handler, $S(l) = (d_1, \mathit{frz}_1)$ and $d_2 \sqsubseteq d_1$ and $d_2 \notin H$ and $d_2 \in Q$.

  By assumption, $S \sqcup_S S'' \neq \top_S$.

  Hence $(S \sqcup_S S'')(l) = (d_1', \mathit{frz}_1')$ where $(d_1, \mathit{frz}_1) \sqsubseteq_p (d_1', \mathit{frz}_1')$.

  By Definition 3.1, $d_1 \sqsubseteq d_1'$.

  By the transitivity of $\sqsubseteq$, $d_2 \sqsubseteq d_1'$.

  Hence $(S \sqcup_S S'')(l) = (d_1', \mathit{frz}_1')$ and $d_2 \sqsubseteq d_1'$ and $d_2 \notin H$ and $d_2 \in Q$.

  Therefore, by E-Spawn-Handler,

  $\langle S \sqcup_S S''; \texttt{freeze}\ l\ \texttt{after}\ Q\ \texttt{with}\ \lambda x.\, e_0, \{e, \dots\}, H\rangle \hookrightarrow$

  $\langle S \sqcup_S S''; \texttt{freeze}\ l\ \texttt{after}\ Q\ \texttt{with}\ \lambda x.\, e_0, \{e_0[x := d_2], e, \dots\}, \{d_2\} \cup H\rangle$,

  as we were required to show.

- Case E-Freeze-Final:

LK: This case wouldn't work but for the $S' \sqcup_S S'' =_{frz} S$ requirement, which makes it a no-op freeze.

Given: $\langle S;\ \texttt{freeze}\ l\ \texttt{after}\ Q\ \texttt{with}\ v, \{v \ldots\}, H\rangle \hookrightarrow \langle S[l \mapsto (d_1, \text{true})];\ d_1\rangle$.

To show: $\langle S \sqcup_S S'';\ \texttt{freeze}\ l\ \texttt{after}\ Q\ \texttt{with}\ v, \{v \ldots\}, H\rangle \hookrightarrow \langle S[l \mapsto (d_1, \text{true})] \sqcup_S S'';\ d_1\rangle$.

We will first show that

$\langle S \sqcup_S S'';\ \texttt{freeze}\ l\ \texttt{after}\ Q\ \texttt{with}\ v, \{v \ldots\}, H\rangle \hookrightarrow \langle (S \sqcup_S S'')[l \mapsto (d_1, \text{true})];\ d_1\rangle$

and then show why this is sufficient.

We proceed by cases on l:

- $l \notin \mathit{dom}(S'')$:

  By assumption, $S[l \mapsto (d_1, \text{true})] \sqcup_S S'' \neq \top_S$.

  By Lemma 3.5, $S \sqsubseteq_S S[l \mapsto (d_1, \text{true})]$.

  Therefore $S \sqcup_S S'' \neq \top_S$.

  Hence, by Definition 3.5, $(S \sqcup_S S'')(l) = S(l)$.

  From the premises of E-Freeze-Final, we have that $S(l) = (d_1, \mathit{frz}_1)$.

  Hence $(S \sqcup_S S'')(l) = (d_1, \mathit{frz}_1)$.

  From the premises of E-Freeze-Final, we have that $\forall d_2\ .\ (d_2 \sqsubseteq d_1 \wedge d_2 \in Q \Rightarrow d_2 \in H)$.

  Therefore, by E-Freeze-Final, we have that

  $\langle S \sqcup_S S'';\ \texttt{freeze}\ l\ \texttt{after}\ Q\ \texttt{with}\ v, \{v \ldots\}, H\rangle \hookrightarrow \langle (S \sqcup_S S'')[l \mapsto (d_1, \text{true})];\ d_1\rangle$.

- $l \in \mathit{dom}(S'')$:

  By assumption, $S[l \mapsto (d_1, \text{true})] \sqcup_S S'' \neq \top_S$.

  By Lemma 3.5, $S \sqsubseteq_S S[l \mapsto (d_1, \text{true})]$.

  Therefore $S \sqcup_S S'' \neq \top_S$.

  Hence, by Definition 3.5, $(S \sqcup_S S'')(l) = S(l) \sqcup_p S''(l)$.

  From the premises of E-Freeze-Final, we have that $S(l) = (d_1, \mathit{frz}_1)$.

  By assumption, $S[l \mapsto (d_1, \text{true})] \sqcup_S S'' =_{frz} S$.

  Therefore $\mathit{frz}_1 = \text{true}$.

Therefore $S(l) = (d_1, \text{true})$.

Therefore $(S \sqcup_S S'')(l) = (d_1, \text{true}) \sqcup_p S''(l)$.

We proceed by cases on $S''(l)$:

* $S''(l) = (d_3, \text{false})$, where $d_3 \sqsubseteq d_1$:

  By Definition 3.2, $(d_1, \text{true}) \sqcup_p (d_3, \text{false}) = (d_1, \text{true})$.

  Therefore $(S \sqcup_S S'')(l) = (d_1, \text{true})$.

  From the premises of E-Freeze-Final, we have that $\forall d_2 \,.\, (d_2 \sqsubseteq d_1 \wedge d_2 \in Q \Rightarrow d_2 \in H)$.

  Therefore, by E-Freeze-Final, we have that

  $\langle S \sqcup_S S''; \texttt{freeze}\, l\, \texttt{after}\, Q\, \texttt{with}\, v, \{v \ldots\}, H\rangle \longrightarrow \langle (S \sqcup_S S'')[l \mapsto (d_1, \text{true})];\, d_1\rangle$. ∎

* $S''(l) = (d_3, \text{false})$, where $d_3 \not\sqsubseteq d_1$:

  By Definition 3.2, $(d_1, \text{true}) \sqcup_p (d_3, \text{false}) = (\top, \text{false})$.

  Therefore $S(l) \sqcup_p S''(l) = (\top, \text{false})$.

  By Definition 3.1, $(\top, \text{false}) = \top_p$.

  Therefore $S(l) \sqcup_p S''(l) = \top_p$.

  Therefore, by Definition 3.5, $S \sqcup_S S'' = \top_S$.

  This is a contradiction.

  Therefore,

  $\langle S \sqcup_S S''; \texttt{freeze}\, l\, \texttt{after}\, Q\, \texttt{with}\, v, \{v \ldots\}, H\rangle \longrightarrow \langle (S \sqcup_S S'')[l \mapsto (d_1, \text{true})];\, d_1\rangle$. ∎

* $S''(l) = (d_3, \text{true})$, where $d_3 = d_1$:

  By Definition 3.2, $(d_1, \text{true}) \sqcup_p (d_3, \text{true}) = (d_1, \text{true})$.

  Therefore $(S \sqcup_S S'')(l) = (d_1, \text{true})$.

  From the premises of E-Freeze-Final, we have that $\forall d_2 \,.\, (d_2 \sqsubseteq d_1 \wedge d_2 \in Q \Rightarrow d_2 \in H)$.

  Therefore, by E-Freeze-Final, we have that

  $\langle S \sqcup_S S''; \texttt{freeze}\, l\, \texttt{after}\, Q\, \texttt{with}\, v, \{v \ldots\}, H\rangle \longrightarrow \langle (S \sqcup_S S'')[l \mapsto (d_1, \text{true})];\, d_1\rangle$. ∎

* $S''(l) = (d_3, \text{true})$, where $d_3 \neq d_1$:

  By Definition 3.2, $(d_1, \text{true}) \sqcup_p (d_3, \text{true}) = (\top, \text{false})$.

  Therefore $S(l) \sqcup_p S''(l) = (\top, \text{false})$.

  By Definition 3.1, $(\top, \text{false}) = \top_p$.

  Therefore $S(l) \sqcup_p S''(l) = \top_p$.

  Therefore, by Definition 3.5, $S \sqcup_S S'' = \top_S$.

  This is a contradiction.

  Therefore,

  $\langle S \sqcup_S S''; \texttt{freeze}\, l\, \texttt{after}\, Q\, \texttt{with}\, v, \{v \ldots\}, H\rangle \hookrightarrow \langle (S \sqcup_S S'')[l \mapsto (d_1, \text{true})];\ d_1\rangle$.∎

In each case we have shown that

$\langle S \sqcup_S S''; \texttt{freeze}\, l\, \texttt{after}\, Q\, \texttt{with}\, v, \{v \ldots\}, H\rangle \hookrightarrow \langle (S \sqcup_S S'')[l \mapsto (d_1, \text{true})];\ d_1\rangle$.

Note that:

$$\begin{aligned}(S \sqcup_S S'')[l \mapsto (d_1, \text{true})] &= S[l \mapsto (d_1, \text{true})] \sqcup_S S''[l \mapsto (d_1, \text{true})] \\ &= S \sqcup_S [l \mapsto (d_1, \text{true})] \sqcup_S S'' \sqcup_S [l \mapsto (d_1, \text{true})] \\ &= S \sqcup_S [l \mapsto (d_1, \text{true})] \sqcup_S S'' \\ &= S[l \mapsto (d_1, \text{true})] \sqcup_S S''.\end{aligned}$$

Therefore

$\langle S \sqcup_S S''; \texttt{freeze}\, l\, \texttt{after}\, Q\, \texttt{with}\, v, \{v \ldots\}, H\rangle \hookrightarrow \langle S[l \mapsto (d_1, \text{true})] \sqcup_S S'';\ d_1\rangle$,

as we were required to show.

- Case E-Freeze-Simple:

  Given: $\langle S; \texttt{freeze}\, l\rangle \hookrightarrow \langle S[l \mapsto (d_1, \text{true})];\ d_1\rangle$.

  To show: $\langle S \sqcup_S S''; \texttt{freeze}\, l\rangle \hookrightarrow \langle S[l \mapsto (d_1, \text{true})] \sqcup_S S'';\ d_1\rangle$.

  We will first show that

  $\langle S \sqcup_S S''; \texttt{freeze}\, l\rangle \hookrightarrow \langle (S \sqcup_S S'')[l \mapsto (d_1, \text{true})];\ d_1\rangle$

and then show why this is sufficient.

We proceed by cases on l:

- $l \notin \mathit{dom}(S'')$:

  By assumption, $S[l \mapsto (d_1, \text{true})] \sqcup_S S'' \neq \top_S$.

  By Lemma 3.5, $S \sqsubseteq_S S[l \mapsto (d_1, \text{true})]$.

  Therefore $S \sqcup_S S'' \neq \top_S$.

  Hence, by Definition 3.5, $(S \sqcup_S S'')(l) = S(l)$.

  From the premise of E-Freeze-Simple, we have that $S(l) = (d_1, \mathit{frz}_1)$.

  Therefore, by E-Freeze-Simple, we have that

  $\langle S \sqcup_S S''; \texttt{freeze}\, l\rangle \hookrightarrow \langle (S \sqcup_S S'')[l \mapsto (d_1, \text{true})];\ d_1\rangle$.

- $l \in \mathit{dom}(S'')$:

  By assumption, $S[l \mapsto (d_1, \text{true})] \sqcup_S S'' \neq \top_S$.

  By Lemma 3.5, $S \sqsubseteq_S S[l \mapsto (d_1, \text{true})]$.

  Therefore $S \sqcup_S S'' \neq \top_S$.

  Hence, by Definition 3.5, $(S \sqcup_S S'')(l) = S(l) \sqcup_p S''(l)$.

  From the premise of E-Freeze-Simple, we have that $S(l) = (d_1, \mathit{frz}_1)$.

  By assumption, $S[l \mapsto (d_1, \text{true})] \sqcup_S S'' =_{\mathit{frz}} S$.

  Therefore $\mathit{frz}_1 = \text{true}$.

  Therefore $(S \sqcup_S S'')(l) = (d_1, \text{true}) \sqcup_p S''(l)$.

  We proceed by cases on $S''(l)$:

  * $S''(l) = (d_2, \text{false})$, where $d_2 \sqsubseteq d_1$:

    By Definition 3.2, $(d_1, \text{true}) \sqcup_p (d_2, \text{false}) = (d_1, \text{true})$.

    Therefore $(S \sqcup_S S'')(l) = (d_1, \text{true})$.

    Therefore, by E-Freeze-Simple, we have that

    $\langle S \sqcup_S S''; \texttt{freeze}\, l\rangle \hookrightarrow \langle (S \sqcup_S S'')[l \mapsto (d_1, \text{true})];\ d_1\rangle$.

  * $S''(l) = (d_2, \text{false})$, where $d_2 \not\sqsubseteq d_1$:

By Definition 3.2, $(d_1, \text{true}) \sqcup_p (d_2, \text{false}) = (\top, \text{false})$.

By Definition 3.1, $(\top, \text{false}) = \top_p$.

Therefore $S(l) \sqcup_p S''(l) = \top_p$.

Therefore, by Definition 3.5, $S \sqcup_S S'' = \top_S$.

This is a contradiction.

Therefore,

$\langle S \sqcup_S S''; \texttt{freeze}\ l\rangle \hookrightarrow \langle (S \sqcup_S S'')[l \mapsto (d_1, \text{true})];\ d_1\rangle$.

* $S''(l) = (d_2, \text{true})$, where $d_2 = d_1$:

  Therefore $(S \sqcup_S S'')(l) = (d_1, \text{true}) \sqcup_p (d_2, \text{true})$.

  By Definition 3.2, $(d_1, \text{true}) \sqcup_p (d_2, \text{true}) = (d_1, \text{true})$.

  Therefore $(S \sqcup_S S'')(l) = (d_1, \text{true})$.

  Therefore, by E-Freeze-Simple, we have that

  $\langle S \sqcup_S S''; \texttt{freeze}\ l\rangle \hookrightarrow \langle (S \sqcup_S S'')[l \mapsto (d_1, \text{true})];\ d_1\rangle$.

* $S''(l) = (d_2, \text{true})$, where $d_2 \neq d_1$:

  By Definition 3.2, $(d_1, \text{true}) \sqcup_p (d_2, \text{true}) = (\top, \text{false})$.

  By Definition 3.1, $(\top, \text{false}) = \top_p$.

  Therefore $S(l) \sqcup_p S''(l) = \top_p$.

  Therefore, by Definition 3.5, $S \sqcup_S S'' = \top_S$.

  This is a contradiction.

  Therefore,

  $\langle S \sqcup_S S''; \texttt{freeze}\ l\rangle \hookrightarrow \langle (S \sqcup_S S'')[l \mapsto (d_1, \text{true})];\ d_1\rangle$.

In each case we have shown that

$\langle S \sqcup_S S''; \texttt{freeze}\ l\rangle \hookrightarrow \langle (S \sqcup_S S'')[l \mapsto (d_1, \text{true})];\ d_1\rangle$.

Note that:

$$\begin{aligned}(S \sqcup_S S'')[l \mapsto (d_1, \text{true})] &= S[l \mapsto (d_1, \text{true})] \sqcup_S S''[l \mapsto (d_1, \text{true})] \\ &= S \sqcup_S [l \mapsto (d_1, \text{true})] \sqcup_S S'' \sqcup_S [l \mapsto (d_1, \text{true})] \\ &= S \sqcup_S [l \mapsto (d_1, \text{true})] \sqcup_S S'' \\ &= S[l \mapsto (d_1, \text{true})] \sqcup_S S''.\end{aligned}$$

Therefore $\langle S \sqcup_S S''; \texttt{freeze}\, l\rangle \hookrightarrow \langle S[l \mapsto (d_1, \text{true})] \sqcup_S S''; d_1\rangle$, as we were required to show.

□

A.15. Proof of Lemma 3.7

Proof. TODO: Needs to be updated.

By induction on the derivation of $\sigma \hookrightarrow \sigma_a$, by cases on the last rule in the derivation.

- E-Eval-Ctxt: TODO: Figure out what to do here.
- E-Beta: $\sigma = \langle S;\ (\lambda x.\, e)\ v\rangle$, and $\sigma_a = \langle S;\ e[x := v]\rangle$.

  Given:

  - $\langle S;\ (\lambda x.\, e)\ v\rangle \hookrightarrow \langle S;\ e[x := v]\rangle$, and
  - $\langle S;\ (\lambda x.\, e)\ v\rangle \hookrightarrow \sigma_b$.

  To show: There exist σ_c, i, j such that

  $\langle S;\ e[x := v]\rangle \hookrightarrow^i \sigma_c$ and $\sigma_b \hookrightarrow^j \sigma_c$ and $i \leq 1$ and $j \leq 1$.

  By inspection of the operational semantics, $\sigma_b = \langle S;\ e[x := v]\rangle$.

  Choose $\sigma_c = \langle S;\ e[x := v]\rangle$, $i = 0$ and $j = 0$.

  Then $\langle S;\ e[x := v]\rangle = \sigma_c$ and $\sigma_b = \sigma_c$, as required.
- E-New: $\sigma = \langle S;\ \texttt{new}\rangle$, and $\sigma_a = \langle S[l \mapsto (\bot, \text{false})];\ l\rangle$.

  Given:

- $\langle S;\ \texttt{new}\rangle \hookrightarrow \langle S[l \mapsto (\bot, \text{false})];\ l\rangle$, and
- $\langle S;\ \texttt{new}\rangle \hookrightarrow \sigma_b$.

To show: There exist σ_c, i, j such that

$\langle S[l \mapsto (\bot, \text{false})];\ l\rangle \hookrightarrow^i \sigma_c$ and $\sigma_b \hookrightarrow^j \sigma_c$ and $i \leq 1$ and $j \leq 1$.

By inspection of the operational semantics, $\sigma_b = \langle S[l' \mapsto (\bot, \text{false})];\ l'\rangle$.

From the side condition of E-New, $l \notin S$.

Therefore, in $\langle S[l' \mapsto (\bot, \text{false})];\ l'\rangle$, we can α-rename l' to l, resulting in $\langle S[l \mapsto (\bot, \text{false})];\ l\rangle$.

Choose $\sigma_c = \langle S[l \mapsto (\bot, \text{false})];\ l\rangle$, $i = 0$ and $j = 0$.

Then $\langle S;\ e[x := v]\rangle = \sigma_c$ and $\sigma_b = \sigma_c$, as required.

- E-Put: $\sigma = \langle S;\ \texttt{put}\ l\ d_2\rangle$, and $\sigma_a = \langle S[l \mapsto p_2];\ \texttt{()}\rangle$.

  Given:

  - $\langle S;\ \texttt{put}\ l\ d_2\rangle \hookrightarrow \langle S[l \mapsto p_2];\ \texttt{()}\rangle$, and
  - $\langle S;\ \texttt{put}\ l\ d_2\rangle \hookrightarrow \sigma_b$.

  To show: There exist σ_c, i, j such that

  $\langle S[l \mapsto p_2];\ \texttt{()}\rangle \hookrightarrow^i \sigma_c$ and $\sigma_b \hookrightarrow^j \sigma_c$ and $i \leq 1$ and $j \leq 1$.

  By inspection of the operational semantics, $\sigma_b = \langle S[l \mapsto p_2];\ \texttt{()}\rangle$.

  Choose $\sigma_c = \langle S[l \mapsto p_2];\ \texttt{()}\rangle$, $i = 0$ and $j = 0$.

  Then $\langle S[l \mapsto p_2];\ \texttt{()}\rangle = \sigma_c$ and $\sigma_b = \sigma_c$, as required.

- E-Put-Err: $\sigma = \langle S;\ \texttt{put}\ l\ d_2\rangle$, and $\sigma_a = \textbf{error}$.

  Given:

  - $\langle S;\ \texttt{put}\ l\ d_2\rangle \hookrightarrow \textbf{error}$, and
  - $\langle S;\ \texttt{put}\ l\ d_2\rangle \hookrightarrow \sigma_b$.

  To show: There exist σ_c, i, j such that

  $\textbf{error} \hookrightarrow^i \sigma_c$ and $\sigma_b \hookrightarrow^j \sigma_c$ and $i \leq 1$ and $j \leq 1$.

  By inspection of the operational semantics, $\sigma_b = \textbf{error}$.

  Choose $\sigma_c = \textbf{error}$, $i = 0$ and $j = 0$.

Then **error** $= \sigma_c$ and $\sigma_b = \sigma_c$, as required.

- E-Get: $\sigma = \langle S;\ \texttt{get}\ l\ P \rangle$, and $\sigma_a = \langle S;\ p_2 \rangle$.

  Given:

  - $\langle S;\ \texttt{get}\ l\ P \rangle \longhookrightarrow \langle S;\ p_2 \rangle$, and
  - $\langle S;\ \texttt{get}\ l\ P \rangle \longhookrightarrow \sigma_b$.

  To show: There exist σ_c, i, j such that

  $\langle S;\ p_2 \rangle \longhookrightarrow^i \sigma_c$ and $\sigma_b \longhookrightarrow^j \sigma_c$ and $i \leq 1$ and $j \leq 1$.

  By inspection of the operational semantics, $\sigma_b = \langle S;\ p_2 \rangle$.

  Choose $\sigma_c = \langle S;\ p_2 \rangle$, $i = 0$ and $j = 0$.

  Then $\langle S;\ p_2 \rangle = \sigma_c$ and $\sigma_b = \sigma_c$, as required.

- E-Freeze-Init: $\sigma = \langle S;\ \texttt{freeze}\, l\, \texttt{after}\, Q\, \texttt{with}\, \lambda x.\, e \rangle$, and $\sigma_a = \langle S;\ \texttt{freeze}\, l\, \texttt{after}\, Q\, \texttt{with}\, \lambda x.\, e, \{\}, \{\} \rangle$.

  Given:

  - $\langle S;\ \texttt{freeze}\, l\, \texttt{after}\, Q\, \texttt{with}\, \lambda x.\, e \rangle \longhookrightarrow \langle S;\ \texttt{freeze}\, l\, \texttt{after}\, Q\, \texttt{with}\, \lambda x.\, e, \{\}, \{\} \rangle$, and
  - $\langle S;\ \texttt{freeze}\, l\, \texttt{after}\, Q\, \texttt{with}\, \lambda x.\, e \rangle \longhookrightarrow \sigma_b$.

  To show: There exist σ_c, i, j such that

  $\langle S;\ \texttt{freeze}\, l\, \texttt{after}\, Q\, \texttt{with}\, \lambda x.\, e, \{\}, \{\} \rangle \longhookrightarrow^i \sigma_c$ and $\sigma_b \longhookrightarrow^j \sigma_c$ and $i \leq 1$ and $j \leq 1$.

  By inspection of the operational semantics, $\sigma_b = \langle S;\ \texttt{freeze}\, l\, \texttt{after}\, Q\, \texttt{with}\, \lambda x.\, e, \{\}, \{\} \rangle$.

  Choose $\sigma_c = \langle S;\ \texttt{freeze}\, l\, \texttt{after}\, Q\, \texttt{with}\, \lambda x.\, e, \{\}, \{\} \rangle$, $i = 0$ and $j = 0$.

  Then $\langle S;\ \texttt{freeze}\, l\, \texttt{after}\, Q\, \texttt{with}\, \lambda x.\, e, \{\}, \{\} \rangle = \sigma_c$ and $\sigma_b = \sigma_c$, as required.

- E-Spawn-Handler: $\sigma = \langle S;\ \texttt{freeze}\, l\, \texttt{after}\, Q\, \texttt{with}\, \lambda x.\, e_0, \{e, \dots\}, H \rangle$, and

  $\sigma_a = \langle S;\ \texttt{freeze}\, l\, \texttt{after}\, Q\, \texttt{with}\, \lambda x.\, e_0, \{e_0[x := d_2], e, \dots\}, \{d_2\} \cup H \rangle$.

  Given:

  - $\langle S;\ \texttt{freeze}\, l\, \texttt{after}\, Q\, \texttt{with}\, \lambda x.\, e_0, \{e, \dots\}, H \rangle \longhookrightarrow$

    $\langle S;\ \texttt{freeze}\, l\, \texttt{after}\, Q\, \texttt{with}\, \lambda x.\, e_0, \{e_0[x := d_2], e, \dots\}, \{d_2\} \cup H \rangle$, and
  - $\langle S;\ \texttt{freeze}\, l\, \texttt{after}\, Q\, \texttt{with}\, \lambda x.\, e_0, \{e, \dots\}, H \rangle \longhookrightarrow \sigma_b$.

  To show: There exist σ_c, i, j such that

$\langle S;\ \texttt{freeze}\ l\ \texttt{after}\ Q\ \texttt{with}\ \lambda x.\, e_0, \{e_0[x := d_2], e, \dots\}, \{d_2\} \cup H\rangle \longhookrightarrow^i \sigma_c$ and $\sigma_b \longhookrightarrow^j \sigma_c$ and $i \leq 1$ and $j \leq 1$.

LK: This time, we could actually get different results on the two sides of the diamond because we could choose a different d_2, or we could run a different expression in $\{e, \dots\}$.

By inspection of the operational semantics, one of the following possibilities must hold:

- $\sigma \longhookrightarrow \sigma_b$ by E-Spawn-Handler:

  Hence $\sigma_b = \langle S;\ \texttt{freeze}\ l\ \texttt{after}\ Q\ \texttt{with}\ \lambda x.\, e_0, \{e_0[x := d_2'], e, \dots\}, \{d_2'\} \cup H\rangle$.

  Here, there are two subcases:

  * $d_2' \neq d_2$:

    Choose:

    $\sigma_c = \langle S;\ \texttt{freeze}\ l\ \texttt{after}\ Q\ \texttt{with}\ \lambda x.\, e_0, \{e_0[x := d_2], e_0[x := d_2'], e, \dots\}, \{d_2, d_2'\} \cup H\rangle$,

    $i = 1$ and $j = 1$.

    To show:

    (1) $\sigma_a \longhookrightarrow \sigma_c$, where

    $\sigma_a = \langle S;\ \texttt{freeze}\ l\ \texttt{after}\ Q\ \texttt{with}\ \lambda x.\, e_0, \{e_0[x := d_2], e, \dots\}, \{d_2\} \cup H\rangle$ and

    $\sigma_c = \langle S;\ \texttt{freeze}\ l\ \texttt{after}\ Q\ \texttt{with}\ \lambda x.\, e_0, \{e_0[x := d_2], e_0[x := d_2'], e, \dots\}, \{d_2, d_2'\} \cup H\rangle$.

    The proof is as follows:

    From the premises of E-Spawn-Handler,

    $S(l) = (d_1, \mathit{frz}_1)$ and $d_2' \sqsubseteq d_1$ and $d_2' \notin H$ and $d_2' \in Q$.

    Therefore, by E-Spawn-Handler,

    $\langle S;\ \texttt{freeze}\ l\ \texttt{after}\ Q\ \texttt{with}\ \lambda x.\, e_0, \{e_0[x := d_2], e, \dots\}, \{d_2\} \cup H\rangle \longhookrightarrow$

    $\langle S;\ \texttt{freeze}\ l\ \texttt{after}\ Q\ \texttt{with}\ \lambda x.\, e_0, \{e_0[x := d_2'], e_0[x := d_2], e, \dots\}, \{d_2'\} \cup \{d_2\} \cup H\rangle$,

which is equivalent to

$\langle S;\ \texttt{freeze}\, l\, \texttt{after}\, Q\, \texttt{with}\, \lambda x.\, e_0, \{e_0[x := d_2], e_0[x := d_2'], e, \dots\}, \{d_2, d_2'\} \cup H\rangle$.

Hence $\sigma_a \hookrightarrow \sigma_c$.

(2) $\sigma_b \hookrightarrow \sigma_c$, where

$\sigma_b = \langle S;\ \texttt{freeze}\, l\, \texttt{after}\, Q\, \texttt{with}\, \lambda x.\, e_0, \{e_0[x := d_2'], e, \dots\}, \{d_2'\} \cup H\rangle$, and

$\sigma_c = \langle S;\ \texttt{freeze}\, l\, \texttt{after}\, Q\, \texttt{with}\, \lambda x.\, e_0, \{e_0[x := d_2], e_0[x := d_2'], e, \dots\}, \{d_2, d_2'\} \cup H\rangle$.

The proof is as follows:

From the premises of E-Spawn-Handler,

$S(l) = (d_1, \mathit{frz}_1)$ and $d_2 \sqsubseteq d_1$ and $d_2 \notin H$ and $d_2 \in Q$.

Therefore, by E-Spawn-Handler,

$\langle S;\ \texttt{freeze}\, l\, \texttt{after}\, Q\, \texttt{with}\, \lambda x.\, e_0, \{e_0[x := d_2'], e, \dots\}, \{d_2'\} \cup H\rangle \hookrightarrow$

$\langle S;\ \texttt{freeze}\ l\ \texttt{after}\ Q\ \texttt{with}\ \lambda x.\, e_0, \{e_0[x := d_2], e_0[x := d_2'], e, \dots\}, \{d_2\} \cup \{d_2'\} \cup H\rangle$,

which is equivalent to

$\langle S;\ \texttt{freeze}\, l\, \texttt{after}\, Q\, \texttt{with}\, \lambda x.\, e_0, \{e_0[x := d_2], e_0[x := d_2'], e, \dots\}, \{d_2, d_2'\} \cup H\rangle$.

Hence $\sigma_b \hookrightarrow \sigma_c$.

$*$ $d_2' = d_2$:

Choose: $\sigma_c = \langle S;\ \texttt{freeze}\, l\, \texttt{after}\, Q\, \texttt{with}\, \lambda x.\, e_0, \{e_0[x := d_2], e, \dots\}, \{d_2\} \cup H\rangle$, $i = 0$ and $j = 0$.

Then $\langle S;\ \texttt{freeze}\, l\, \texttt{after}\, Q\, \texttt{with}\, \lambda x.\, e_0, \{e_0[x := d_2], e, \dots\}, \{d_2\} \cup H\rangle = \sigma_c$ and $\sigma_b = \sigma_c$, as required.

– $\sigma \hookrightarrow \sigma_b$ by E-Eval-Ctxt:

Hence $\sigma_b = \langle S'; \texttt{freeze}\ l\ \texttt{after}\ Q\ \texttt{with}\ \lambda x.\, e_0, \{e', \dots\}, H\rangle$.

LK: Minor cheat here: the expression that could step could be *any* expression in the set $\{e, \dots\}$, not necessarily the first one. I'm hoping it's enough to just write out this case.

Choose:

$\sigma_c = \langle S'; \texttt{freeze}\ l\ \texttt{after}\ Q\ \texttt{with}\ \lambda x.\, e_0, \{e_0[x := d_2], e', \dots\}, \{d_2\} \cup H\rangle$,

$i = 1$ and $j = 1$.

To show:

(1) $\sigma_a \hookrightarrow \sigma_c$, where

$\sigma_a = \langle S; \texttt{freeze}\ l\ \texttt{after}\ Q\ \texttt{with}\ \lambda x.\, e_0, \{e_0[x := d_2], e, \dots\}, \{d_2\} \cup H\rangle$ and

$\sigma_c = \langle S'; \texttt{freeze}\ l\ \texttt{after}\ Q\ \texttt{with}\ \lambda x.\, e_0, \{e_0[x := d_2], e', \dots\}, \{d_2\} \cup H\rangle$.

The proof is as follows:

From the premise of E-Eval-Ctxt, $\langle S; e\rangle \hookrightarrow \langle S'; e'\rangle$.

From the definition of evaluation contexts,

$\texttt{freeze}\ l\ \texttt{after}\ Q\ \texttt{with}\ \lambda x.\, e_0, \{e_0[x := d_2], [\,], \dots\}, \{d_2\} \cup H$

is an evaluation context.

Hence, by E-Eval-Ctxt,

$\langle S; \texttt{freeze}\ l\ \texttt{after}\ Q\ \texttt{with}\ \lambda x.\, e_0, \{e_0[x := d_2], e, \dots\}, \{d_2\} \cup H\rangle \hookrightarrow$

$\langle S'; \texttt{freeze}\ l\ \texttt{after}\ Q\ \texttt{with}\ \lambda x.\, e_0, \{e_0[x := d_2], e', \dots\}, \{d_2\} \cup H\rangle$.

Hence $\sigma_a \hookrightarrow \sigma_c$.

(2) $\sigma_b \hookrightarrow \sigma_c$, where

$\sigma_b = \langle S'; \texttt{freeze}\ l\ \texttt{after}\ Q\ \texttt{with}\ \lambda x.\, e_0, \{e', \dots\}, H\rangle$, and

$\sigma_c = \langle S'; \texttt{freeze}\ l\ \texttt{after}\ Q\ \texttt{with}\ \lambda x.\, e_0, \{e_0[x := d_2], e', \dots\}, \{d_2\} \cup H\rangle$.

The proof is as follows:

From the premises of E-Spawn-Handler,

$S(l) = (d_1, \mathit{frz}_1)$ and $d_2 \sqsubseteq d_1$ and $d_2 \notin H$ and $d_2 \in Q$.

Hence, by E-Spawn-Handler,

$\langle S';\ \texttt{freeze}\ l\ \texttt{after}\ Q\ \texttt{with}\ \lambda x.\, e_0, \{e', \dots\}, H\rangle \longhookrightarrow$

$\langle S';\ \texttt{freeze}\ l\ \texttt{after}\ Q\ \texttt{with}\ \lambda x.\, e_0, \{e_0[x := d_2], e', \dots\}, \{d_2\} \cup H\rangle$.

Hence $\sigma_b \longhookrightarrow \sigma_c$.

- E-Freeze-Final: $\sigma = \langle S;\ \texttt{freeze}\, l\, \texttt{after}\, Q\, \texttt{with}\, v, \{v \dots\}, H\rangle$, and $\sigma_a = \langle S[l \mapsto (d_1, \text{true})];\ d_1\rangle$.

  Given:

  - $\langle S;\ \texttt{freeze}\ l\ \texttt{after}\ Q\ \texttt{with}\ v, \{v \dots\}, H\rangle \longhookrightarrow \langle S[l \mapsto (d_1, \text{true})];\ d_1\rangle$, and
  - $\langle S;\ \texttt{freeze}\ l\ \texttt{after}\ Q\ \texttt{with}\ v, \{v \dots\}, H\rangle \longhookrightarrow \sigma_b$.

  To show: There exist σ_c, i, j such that

  $\langle S[l \mapsto (d_1, \text{true})];\ d_1\rangle \longhookrightarrow^i \sigma_c$ and $\sigma_b \longhookrightarrow^j \sigma_c$ and $i \leq 1$ and $j \leq 1$.

  By inspection of the operational semantics, $\sigma_b = \langle S[l \mapsto (d_1, \text{true})];\ d_1\rangle$.

  Choose $\sigma_c = \langle S[l \mapsto (d_1, \text{true})];\ d_1\rangle$, $i = 0$ and $j = 0$.

  Then $\langle S[l \mapsto (d_1, \text{true})];\ d_1\rangle = \sigma_c$ and $\sigma_b = \sigma_c$, as required.

- E-Freeze-Simple: $\sigma = \langle S;\ \texttt{freeze}\ l\rangle$, and $\sigma_a = \langle S[l \mapsto (d_1, \text{true})];\ d_1\rangle$.

  Given:

  - $\langle S;\ \texttt{freeze}\ l\rangle \longhookrightarrow \langle S[l \mapsto (d_1, \text{true})];\ d_1\rangle$, and
  - $\langle S;\ \texttt{freeze}\ l\rangle \longhookrightarrow \sigma_b$.

  To show: There exist σ_c, i, j such that $\langle S[l \mapsto (d_1, \text{true})];\ d_1\rangle \longhookrightarrow^i \sigma_c$ and $\sigma_b \longhookrightarrow^j \sigma_c$ and $i \leq 1$ and $j \leq 1$.

  By inspection of the operational semantics, $\sigma_b = \langle S[l \mapsto (d_1, \text{true})];\ d_1\rangle$.

  Choose $\sigma_c = \langle S[l \mapsto (d_1, \text{true})];\ d_1\rangle$, $i = 0$ and $j = 0$.

  Then $\langle S[l \mapsto (d_1, \text{true})];\ d_1\rangle = \sigma_c$ and $\sigma_b = \sigma_c$, as required.

□

A.16. Proof of Lemma 3.8

Proof. TODO: Revise to mention permutations.

We proceed by induction on m. In the base case of $m = 1$, the result is immediate from Lemma 3.7, with $k = 1$.

For the induction step, suppose $\sigma \longmapsto^m \sigma'' \longmapsto \sigma'''$ and suppose the lemma holds for m.

We show that it holds for $m + 1$, as follows.

We are required to show that either:

(1) there exist σ_c, i, j such that $\sigma' \longmapsto^i \sigma_c$ and $\sigma''' \longmapsto^j \sigma_c$ and $i \leq m + 1$ and $j \leq 1$, or

(2) there exists $k \leq m + 1$ such that $\sigma' \longmapsto^k$ **error**, or there exists $k \leq 1$ such that $\sigma''' \longmapsto^k$ **error**.

From the induction hypothesis, we have that either:

(1) there exist σ'_c, i', j' such that $\sigma' \longmapsto^{i'} \sigma'_c$ and $\sigma'' \longmapsto^{j'} \sigma'_c$ and $i' \leq m$ and $j' \leq 1$, or

(2) there exists $k' \leq m$ such that $\sigma' \longmapsto^{k'}$ **error**, or there exists $k' \leq 1$ such that $\sigma'' \longmapsto^{k'}$ **error**.

We consider these two cases in turn:

(1) There exist σ'_c, i', j' such that $\sigma' \longmapsto^{i'} \sigma'_c$ and $\sigma'' \longmapsto^{j'} \sigma'_c$ and $i' \leq m$ and $j' \leq 1$:

LK: This gets very hairy. I could spell things out in more detail, but it's MUCH better explained with a drawing, which I'm happy to do later. :)

We proceed by cases on j':

- If $j' = 0$, then $\sigma'' = \sigma'_c$. We can then choose $\sigma_c = \sigma'''$ and $i = i' + 1$ and $j = 0$.
- If $j' = 1$:

  From $\sigma'' \longmapsto \sigma'''$ and $\sigma'' \longmapsto^{j'} \sigma'_c$ and Lemma 3.7, one of the following two cases is true:

  (a) There exist σ''_c and i'' and j'' such that $\sigma'_c \longmapsto^{i''} \sigma''_c$ and $\sigma''' \longmapsto^{j''} \sigma''_c$ and $i'' \leq 1$ and $j'' \leq 1$. So we also have $\sigma' \longmapsto^{i'} \sigma'_c \longmapsto^{i''} \sigma''_c$. In summary, we pick $\sigma_c = \sigma''_c$ and $i = i' + i''$ and $j = j''$, which is sufficient because $i = i' + i'' \leq m + 1$ and $j = j'' \leq 1$.

(b) $\sigma''' \longmapsto$ **error** or $\sigma'_c \longmapsto$ **error**.

If $\sigma''' \longmapsto$ **error**, then choosing $k = 1$ satisfies the proof.

Otherwise, $\sigma'_c \longmapsto$ **error**, therefore $\sigma' \longmapsto^{i'} \sigma'_c \longmapsto$ **error**.

Hence $\sigma' \longmapsto^{i'+1}$ **error**.

Since $i' \leq m$, we have that $i' + 1 \leq m + 1$, and so choosing $k = i' + 1$ satisfies the proof.

(2) There exists $k' \leq m$ such that $\sigma' \longmapsto^{k'}$ **error**, or there exists $k' \leq 1$ such that $\sigma'' \longmapsto^{k'}$ **error**:

If there exists $k' \leq m$ such that $\sigma' \longmapsto^{k'}$ **error**, then choosing $k = k'$ satisfies the proof.

Otherwise, there exists $k' \leq 1$ such that $\sigma'' \longmapsto^{k'}$ **error**. We proceed by cases on k':

- If $k' = 0$, then $\sigma'' =$ **error**.

  Hence this case is not possible, since $\sigma'' \longmapsto \sigma'''$ and **error** cannot step.

- If $k' = 1$:

  From $\sigma'' \longmapsto \sigma'''$ and $\sigma'' \longmapsto^{k'}$ **error** and Lemma 3.7, one of the following two cases is true:

  (a) There exist σ''_c and i'' and j'' such that **error** $\longmapsto^{i''} \sigma''_c$ and $\sigma''' \longmapsto^{j''} \sigma''_c$ and $i'' \leq 1$ and $j'' \leq 1$.

  Since **error** cannot step, $i'' = 0$ and $\sigma''_c =$ **error**.

  Hence $\sigma''' \longmapsto^{j''}$ **error**.

  LK: This is the one place that we need to allow k to be ≤ 1 instead of exactly 1.

  Since $j'' \leq 1$, choosing $k = j''$ satisfies the proof.

  (b) **error** $\longmapsto$ **error** or $\sigma''' \longmapsto$ **error**.

  Since **error** cannot step, $\sigma''' \longmapsto$ **error**.

  Hence choosing $k = 1$ satisfies the proof.

□

A.17. Proof of Lemma 3.9

Proof. TODO: Revise to mention permutations.

We proceed by induction on n. In the base case of $n = 1$, the result is immediate from Lemma 3.8.

For the induction step, suppose $\sigma \longmapsto^n \sigma' \longmapsto \sigma'''$ and suppose the lemma holds for n.

We show that it holds for $n + 1$, as follows.

We are required to show that either:

(1) there exist σ_c, i, j such that $\sigma''' \longmapsto^i \sigma_c$ and $\sigma'' \longmapsto^j \sigma_c$ and $i \leq m$ and $j \leq n + 1$, or

(2) there exists $k \leq m$ such that $\sigma''' \longmapsto^k$ **error**, or there exists $k \leq n + 1$ such that $\sigma'' \longmapsto^k$ **error**.

From the induction hypothesis, we have that either:

(1) there exist σ'_c, i', j' such that $\sigma' \longmapsto^{i'} \sigma'_c$ and $\sigma'' \longmapsto^{j'} \sigma'_c$ and $i' \leq m$ and $j' \leq n$, or

(2) there exists $k' \leq m$ such that $\sigma' \longmapsto^{k'}$ **error**, or there exists $k' \leq n$ such that $\sigma'' \longmapsto^{k'}$ **error**.

We consider these two cases in turn:

(1) There exist σ'_c, i', j' such that $\sigma' \longmapsto^{i'} \sigma'_c$ and $\sigma'' \longmapsto^{j'} \sigma'_c$ and $i' \leq m$ and $j' \leq n$:

We proceed by cases on i':

- If $i' = 0$, then $\sigma' = \sigma'_c$. We can then choose $\sigma_c = \sigma'''$ and $i = 0$ and $j = j' + 1$.
- If $i' \geq 1$:

  From $\sigma' \longmapsto \sigma'''$ and $\sigma' \longmapsto^{i'} \sigma'_c$ and Lemma 3.8, one of the following two cases is true:

  (a) There exist σ''_c and i'' and j'' such that $\sigma''' \longmapsto^{i''} \sigma''_c$ and $\sigma'_c \longmapsto^{j''} \sigma''_c$ and $i'' \leq i'$ and $j'' \leq 1$. So we also have $\sigma'' \longmapsto^{j'} \sigma'_c \longmapsto^{j''} \sigma''_c$. In summary, we pick $\sigma_c = \sigma''_c$ and $i = i''$ and $j = j' + j''$, which is sufficient because $i = i'' \leq i' \leq m$ and $j = j' + j'' \leq n + 1$.

  (b) There exists $k'' \leq i'$ such that $\sigma''' \longmapsto^{k''}$ **error**, or there exists $k'' \leq 1$ such that $\sigma'_c \longmapsto^{k''}$ **error**.

If there exists $k'' \leq i'$ such that $\sigma''' \longmapsto^{k''}$ **error**, then choosing $k = k''$ satisfies the proof, since $k'' \leq i' \leq m$.

Otherwise, there exists $k'' \leq 1$ such that $\sigma'_c \longmapsto^{k''}$ **error**.

Therefore, $\sigma'' \longmapsto^{j'} \sigma'_c \longmapsto^{k''}$ **error**.

Hence $\sigma'' \longmapsto^{j'+k''}$ **error**.

Since $j' \leq n$ and $k'' \leq 1$, $j' + k'' \leq n + 1$.

Hence choosing $k = j' + k''$ satisfies the proof.

(2) There exists $k' \leq m$ such that $\sigma' \longmapsto^{k'}$ **error**, or there exists $k' \leq n$ such that $\sigma'' \longmapsto^{k'}$ **error**:

If there exists $k' \leq n$ such that $\sigma'' \longmapsto^{k'}$ **error**, then choosing $k = k'$ satisfies the proof.

Otherwise, there exists $k' \leq m$ such that $\sigma' \longmapsto^{k'}$ **error**. We proceed by cases on k':

- If $k' = 0$, then $\sigma' =$ **error**.

  Hence this case is not possible, since $\sigma' \longmapsto \sigma'''$ and **error** cannot step.

- If $k' \geq 1$:

  From $\sigma' \longmapsto \sigma'''$ and $\sigma' \longmapsto^{k'}$ **error** and Lemma 3.8, one of the following two cases is true:

  (a) There exist σ''_c and i'' and j'' such that $\sigma''' \longmapsto^{i''} \sigma''_c$ and **error** $\longmapsto^{j''} \sigma''_c$ and $i'' \leq k'$ and $j'' \leq 1$.

  Since **error** cannot step, $j'' = 0$ and $\sigma''_c =$ **error**.

  Hence $\sigma''' \longmapsto^{i''}$ **error**.

  Since $i'' \leq k' \leq m$, choosing $k = i''$ satisfies the proof.

  (b) There exists $k'' \leq k'$ such that $\sigma''' \longmapsto^{k''}$ **error**, or there exists $k'' \leq 1$ such that **error** $\longmapsto^{k''}$ **error**.

  Since **error** cannot step, there exists $k'' \leq k'$ such that $\sigma''' \longmapsto^{k''}$ **error**.

  Since $k'' \leq k' \leq m$, choosing $k = k''$ satisfies the proof.

□

A.18. Proof of Theorem 5.2

Proof. Consider replica i of a threshold CvRDT $(S, \leq, s^0, q, t, u, m)$. Let $\mathcal{S}$ be a threshold set with respect to $(S, \leq)$. Consider a method execution $t_i^{k+1}(\mathcal{S})$ (*i.e.*, a threshold query that is the $k+1$th method execution on replica i, with threshold set $\mathcal{S}$ as its argument) that returns some set of activation states $S_a \in \mathcal{S}$.

For part 1 of the theorem, we have to show that threshold queries with $\mathcal{S}$ as their argument will always return S_a on subsequent executions at i. That is, we have to show that, for all $k' > (k+1)$, the threshold query $t_i^{k'}(\mathcal{S})$ on i returns S_a.

Since $t_i^{k+1}(\mathcal{S})$ returns S_a, from Definition 5.4 we have that for some activation state $s_a \in S_a$, the condition $s_a \leq s_i^k$ holds. Consider arbitrary $k' > (k+1)$. Since state is inflationary across updates, we know that the state $s_i^{k'}$ after method execution k' is at least s_i^k. That is, $s_i^k \leq s_i^{k'}$. By transitivity of $\leq$, then, $s_a \leq s_i^{k'}$. Hence, by Definition 5.4, $t_i^{k'}(\mathcal{S})$ returns S_a.

For part 2 of the theorem, consider some replica j of $(S, \leq, s^0, q, t, u, m)$, located at process p_j. We are required to show that, for all $x \geq 0$, the threshold query $t_j^{x+1}(\mathcal{S})$ returns S_a eventually, and blocks until it does.[1] That is, we must show that, for all $x \geq 0$, there exists some finite $n \geq 0$ such that

- for all i in the range $0 \leq i \leq n-1$, the threshold query $t_j^{x+1+i}(\mathcal{S})$ returns block, and
- for all $i \geq n$, the threshold query $t_j^{x+1+i}(\mathcal{S})$ returns S_a.

Consider arbitrary $x \geq 0$. Recall that s_j^x is the state of replica j after the xth method execution, and therefore s_j^x is also the state of j when $t_j^{x+1}(\mathcal{S})$ runs. We have three cases to consider:

- $s_i^k \leq s_j^x$. (That is, replica i's state after the kth method execution on i is *at or below* replica j's state after the xth method execution on j.) Choose $n = 0$. We have to show that, for all $i \geq n$, the threshold query $t_j^{x+1+i}(\mathcal{S})$ returns S_a. Since $t_i^{k+1}(\mathcal{S})$ returns S_a, we know that there exists an

[1] The occurrences of $k+1$ and $x+1$ in this proof are an artifact of how we index method executions starting from 1, but states starting from 0. The initial state (of every replica) is s^0, and so s_i^k is the state of replica i after method execution k has completed at i.

$s_a \in S_a$ such that $s_a \leq s_i^k$. Since $s_i^k \leq s_j^x$, we have by transitivity of $\leq$ that $s_a \leq s_j^x$. Therefore, by Definition 5.4, $t_j^{x+1}(\mathcal{S})$ returns S_a. Then, by part 1 of the theorem, we have that subsequent executions $t_j^{x+1+i}(\mathcal{S})$ at replica j will also return S_a, and so the case holds. (Note that this case includes the possibility $s_i^k \equiv s^0$, in which no updates have executed at replica i.)

- $s_i^k > s_j^x$. (That is, replica i's state after the kth method execution on i is *above* replica j's state after the xth method execution on j.)

  We have two subcases:

  - There exists some activation state $s'_a \in S_a$ for which $s'_a \leq s_j^x$. In this case, we choose $n = 0$. We have to show that, for all $i \geq n$, the threshold query $t_j^{x+1+i}(\mathcal{S})$ returns S_a. Since $s'_a \leq s_j^x$, by Definition 5.4, $t_j^{x+1}(\mathcal{S})$ returns S_a. Then, by part 1 of the theorem, we have that subsequent executions $t_j^{x+1+i}(\mathcal{S})$ at replica j will also return S_a, and so the case holds.
  - There is no activation state $s'_a \in S_a$ for which $s'_a \leq s_j^x$. Since $t_i^{k+1}(\mathcal{S})$ returns S_a, we know that there is some update $u_i^{k'}(a)$ in i's causal history, for some $k' < (k+1)$, that updates i from a state at or below s_j^x to s_i^k.[2] By eventual delivery, $u_i^{k'}(a)$ is eventually delivered at j. Hence some update or updates that will increase j's state from s_j^x to a state at or above some s'_a must reach replica j.[3]

    Let the $x+1+r$th method execution on j be the first update on j that updates its state to some $s_j^{x+1+r} \geq s'_a$, for some activation state $s'_a \in S_a$. Choose $n = r+1$. We have to show that, for all i in the range $0 \leq i \leq r$, the threshold query $t_j^{x+1+i}(\mathcal{S})$ returns block, and that for all $i \geq r+1$, the threshold query $t_j^{x+1+i}(\mathcal{S})$ returns S_a.

    For the former, since the $x+1+r$th method execution on j is the first one that updates its state to $s_j^{x+1+r} \geq s'_a$, we have by Definition 5.4 that for all i in the range $0 \leq i \leq r$, the threshold query $t_j^{x+1+i}(\mathcal{S})$ returns block.

[2] We know that i's state was once at or below s_j^x, because i and j started at the same state s^0 and can both only grow. Hence the least that s_j^x can be is s^0, and we know that i was originally s^0 as well.

[3] We say "some update or updates" because the exact update $u_i^{k'}(a)$ may not be the update that causes the threshold query at j to unblock; a different update or updates could do it. Nevertheless, the existence of $u_i^{k'}(a)$ means that there is at least one update that will suffice to unblock the threshold query.

For the latter, since $s_j^{x+1+r} \geq s'_a$, by Definition 5.4 we have that $t_j^{x+1+r+1}(\mathcal{S})$ returns S_a, and by part 1 of the theorem, we have that for $i \geq r+1$, subsequent executions $t_j^{x+1+i}(\mathcal{S})$ at replica j will also return S_a, and so the case holds.

- $s_i^k \not\leq s_j^x$ and $s_j^x \not\leq s_i^k$. (That is, replica i's state after the kth method execution on i is *not comparable* to replica j's state after the xth method execution on j.) Similar to the previous case.

□

APPENDIX B

PLT Redex Models of λ_{LVar} and λ_{LVish}

TODO: Edit this text.

We have developed a runnable version of the LVish calculus[1] using the PLT Redex semantics engineering toolkit [20]. In the Redex of today, it is not possible to directly parameterize a language definition by a lattice.[2] Instead, taking advantage of Racket's syntactic abstraction capabilities, we define a Racket macro, `define-LVish-language`, that wraps a template implementing the lattice-agnostic semantics of λ_{LVish}, and takes the following arguments:

- a *name*, which becomes the *lang-name* passed to Redex's `define-language` form;
- a *"downset" operation*, a Racket-level procedure that takes a lattice element and returns the (finite) set of all lattice elements that are below that element (this operation is used to implement the semantics of `freeze` — `after` — `with`, in particular, to determine when the E-Freeze-Final rule can fire);
- a *lub operation*, a Racket-level procedure that takes two lattice elements and returns a lattice element; and
- a (possibly infinite) set of *lattice elements* represented as Redex *patterns*.

Given these arguments, `define-LVish-language` generates a Redex model specialized to the application-specific lattice in question. For instance, to instantiate a model called `nat`, where the application-specific lattice is the natural numbers with `max` as the least upper bound, one writes:

```
(define-LVish-language nat downset-op max natural)
```

[1] Available at http://github.com/iu-parfunc/lvars.
[2] See discussion at http://lists.racket-lang.org/users/archive/2013-April/057075.html.

where `downset-op` is separately defined. Here, `downset-op` and `max` are Racket procedures. `natural` is a Redex pattern that has no meaning to Racket proper, but because `define-LVish-language` is a macro, `natural` is not evaluated until it is in the context of Redex.LK: This might be too much information, or a little confusing. It's a nice illustration of the power of macros, though. I'd welcome suggestions for how to word it differently.

TODO: Freshen up the Redex models of λ_{LVar} and λ_{LVish} and add them here.