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: Some kind of introductory paragraph...

Dan Friedman invited me to be a teaching assistant for his programming languages course, known as C311 at IU, in spring 2009, which eventually led to my first research project in grad school: working on miniKanren\(^1\) over summer 2009 with Dan, Will Byrd, and a group of relational programming aficionados fueled by intense curiosity and Mother Bear’s pizza. I was in way over my head, but we had a great time, and I went on to help out with C311 for two more semesters under Dan and Will’s expert guidance.

Amal Ahmed joined the IU faculty in 2009 and was a tremendous influence on me as I began truly learning how to read PL research papers and, eventually, write them. Although we never worked on determinism as such, it was nevertheless from Amal that I learned how to design and prove properties of core calculi much like the $\lambda_{\text{LVar}}$ and $\lambda_{\text{LVish}}$ calculi in this dissertation. Beyond that, Amal had a huge influence on the culture of the PL group at IU. Weekly “PL Wonks” meetings are a given these days, but it wasn’t so very long ago that the PL group didn’t have particularly regular meetings, or even much of a group identity. It was Amal who gathered us all in a room and told us that from then on, we were all going to give talks every semester. It was also at Amal’s urging that I attended my first PL conference, ICFP 2010 in Baltimore. I don’t expect any of them to remember it, but that conference was where I first met an astonishing number of the people who have become my mentors, colleagues, and dear friends in the PL community, including Neel Krishnaswami, Chung-chieh Shan, Tim Chevalier, Rob Simmons, Jason Reed, Ron Garcia, Stevie Strickland, Dan Licata, and Chris Martens.

When Amal left IU for Northeastern in 2011, I had to make the difficult decision between going with her or staying, and although I chose to stay at IU and work with Ryan Newton on what would eventually become this dissertation, I think that her influence on the work I ended up doing is evident. In fact, it

\(^1\)http://www.minikanren.org — not that the website existed at the time!
was Amal who originally pointed out the similarity between the Independence Lemma and the frame rule that I discuss in Section 2.5.5.

Ryan Newton, my advisor, runs at a faster clock speed than most people. His first act as a new faculty member at IU in fall 2011 was to arrange for our department to get a fancy espresso machine — and to kick-start a community of student espresso enthusiasts who lovingly maintain it. In fact, now that I think about it, I’m not sure it was a coincidence that Ryan had the espresso machine installed in the kitchen directly next to my office... Anyway, I took Ryan’s seminar course that fall, which was how I started learning about deterministic parallel programming models. In November 2011, we started discussing the idea of generalizing single-assignment models — we got so caught up in talking about it one morning that I actually made him late for lunch with our visiting colloquium speaker, Bjarne Stroustrup — and Ryan suggested that I could help him write an NSF grant proposal to continue working on the idea. We somehow managed to get our proposal together as 2011 came to a close, and to my astonishment, the proposal was funded. I gratefully acknowledge everyone at the NSF who took a chance on grant CCF-1218375 — without that early vote of confidence, I’m not sure I would have been able to keep my spirits up during the year that followed, in which it took us four attempts to get our first LVars paper published. (In retrospect, I am also deeply grateful to the anonymous reviewers of POPL 2013, ESOP 2013, and ICFP 2013, whose advice helped us turn LVars from a half-baked idea into a convincing research contribution.)

In fall 2012, I got in touch with Aaron Turon (whom I’d previously met in 2011 while on a visit to Northwestern) and Neel Krishnaswami. I had become interested in separation logic as a result of working on LVars, and wanted to come up with a separation logic for deterministic parallelism — but I couldn’t do it by myself, and who better to collaborate with than Aaron and Neel? When we all met up in Saarbrücken in January 2013, though, Aaron and Neel wanted to first focus on making the LVars programming model more expressive. That train of thought eventually took us all the way to event handlers, quiescence,
freezing, and quasi-determinism, the topics of Chapter 3, and to our POPL 2014 paper. It also led to the LVish Haskell library (Chapter 4), which was mostly the work of Aaron and Ryan. In other words, this dissertation as it is today could not have existed without the contributions of Aaron and Neel. I’m thankful also to Derek Dreyer for helping facilitate our collaboration, as well as for giving me an excuse to give a *Big Lebowski*-themed talk at MPI-SWS.

Before we managed to publish anything, though, I had started to hear from people in the distributed systems community who were interested in LVars. Pursuing the connection between LVars and distributed data consistency — the topic of Chapter 5 — has been one of the most exciting and rewarding parts of this whole journey for me, largely because of all the distributed systems researchers who have met me more than halfway. I especially want to thank the BOOM group at Berkeley, particularly Joe Hellerstein and his students Peter Alvaro, Peter Bailis, and Neil Conway; and the fine folks from Basho Technologies, particularly Sam Elliott and Chris Meiklejohn. Speaking about LVars at RICON 2013 was one of the highlights of my Ph.D. experience; thanks to everyone at Basho who made it possible.

TODO: Lots more people to acknowledge: Amr Sabry; Larry Moss; Chung-chieh Shan; Mozilla Research folks, especially Dave Herman; The original Rust interns: Tim Chevalier, Michael “Sully” Sullivan, Paul Stansifer, Eric Holk; Sam Tobin-Hochstadt; PL Wonks; and probably some others I’m forgetting.

Finally, this dissertation is dedicated to my husband, Alex Rudnick, without whose unflagging love, support, advice, and encouragement I doubt I would have started grad school, let alone finished.
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 a given 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

1.1. The deterministic-by-construction parallel programming landscape 2
1.2. Monotonic data structures as a basis for deterministic parallelism 4
1.3. Quasi-deterministic and event-driven programming with LVars 5
1.4. The LVish library 7
1.5. Deterministic threshold queries of distributed data structures 8
1.6. Thesis statement, and organization of the rest of this dissertation 9
1.7. Previously published work 10

Chapter 2. LVars: lattice-based data structures for deterministic parallelism 13

2.1. Motivating example: a parallel, pipelined graph computation 15
2.2. LVars by example 19
2.3. Lattices, stores, and determinism 23
2.4. \( \lambda_{\text{LVars}} \): syntax and semantics 28
2.5. Proof of determinism for \( \lambda_{\text{LVars}} \) 33
2.6. Generalizing the \texttt{put} and \texttt{get} operations 44

Chapter 3. Quasi-deterministic and event-driven programming with LVars 55

3.1. LVish, informally 57
3.2. LVish, formally 61
3.3. Proof of quasi-determinism for \( \lambda_{\text{LVish}} \) 73

Chapter 4. The LVish library: interface, implementation, and evaluation 85

4.1. The big picture 86
4.2. The LVish library interface for application writers 87
4.3. Par-monad transformers and disjoint parallel update 97
4.4. The LVish library implementation 103
4.5. Case study: parallelizing $k$-CFA with LVish 113
4.6. Case study: parallelizing PhyBin with LVish 120

Chapter 5. Deterministic threshold queries of distributed data structures 125
5.1. Background: CvRDTs and eventual consistency 127
5.2. Adding threshold queries to CvRDTs 130
5.3. Determinism of threshold queries 134

Chapter 6. Related work 137
6.1. Deterministic Parallel Java (DPJ) 138
6.2. FlowPools 138
6.3. Concurrent Revisions 139
6.4. Conflict-free replicated data types 140
6.5. Bloom and Bloom$^L$ 140
6.6. Frame properties and separation logics 141

Chapter 7. Summary and future work 143
7.1. Remapping the deterministic parallel landscape 143
7.2. Distributed programming and the future of LVars and LVish 145

Bibliography 147

Appendix A. Proofs 151
A.1. Proof of Lemma 2.1 151
A.2. Proof of Lemma 2.2 154
A.3. Proof of Lemma 2.3 155
A.4. Proof of Lemma 2.4 155
A.5. Proof of Lemma 2.5 157
A.6. Proof of Lemma 2.6 161
A.7. Proof of Lemma 2.8 164
A.8. Proof of Lemma 2.9
A.9. Proof of Lemma 2.10
A.10. Proof of Lemma 3.2
A.11. Proof of Lemma 3.3
A.12. Proof of Lemma 3.4
A.13. Proof of Lemma 3.5
A.14. Proof of Lemma 3.6
A.15. Proof of Lemma 3.7
A.16. Proof of Lemma 3.8
A.17. Proof of Lemma 3.10
A.18. Proof of Lemma 3.11
A.20. Proof of Theorem 5.2

Appendix B. PLT Redex Models of $\lambda_{\text{Var}}$ and $\lambda_{\text{Vish}}$
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. Although there are many ways to construct individual deterministic programs and verify their determinism, deterministic-by-construction programming models provide a language-level guarantee of determinism that holds for all programs written using the model.

A deterministic program is one that has the same observable behavior every time it is 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 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. But even with such a limited notion of what is observable, schedule nondeterminism can affect 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 [38]. The key characteristic of this style of programming is lack of side effects: because programs do not 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 [49, 14] or the River Trail API for JavaScript [28], each of which extend existing languages with *parallel array* data types and (observably) pure operations on them.

- **Data-flow parallelism.** In *Kahn process networks* (KPNs) [32], as well as in the more restricted *synchronous data flow* systems [35], a network of independent “computing stations” communicate with each other through first-in, first-out (FIFO) queues, or *channels*. In this model, each computing station is a task, and channels are the only means of sharing state between tasks. Furthermore, reading data from a channel 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 [27].
1. INTRODUCTION

- **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 [51]; 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 of IVars incorporate them into already-nondeterministic programming environments, the monad-par Haskell library [39] 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. Sophisticated permissions systems and type systems can 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 (DPI) [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 want to exploit data-flow pipeline parallelism via FIFO queues, but we also want to mutate disjoint slices 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—and if we do manage to compose them, it is not obvious whether 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. 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, and for which the timing of updates is not observable—emerge as a common theme of guaranteed-deterministic programming models.

In this dissertation, I show that lattice-based data structures, or LVars, offer a general approach to deterministic parallel programming that takes monotonicity as a starting point. The states an LVar can take on are elements of a given lattice. This 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 make the state of an LVar “grow” with respect to the lattice, because it updates the LVar to the least upper bound of the current state and the new state. For example, on LVars of collection type, such as sets, the put operation typically inserts an element.
- The get operation allows only limited observations of the contents of an LVar. It requires the user to specify a threshold set of minimum values that can be read from the LVar, where every two elements in the threshold set must have the lattice’s greatest element \( \top \) as their least upper bound. A call to get blocks until the LVar in question reaches a (unique) value in the threshold set, then unblocks and returns that value.

Together, least-upper-bound writes via put and threshold reads via get yield a programming model that is deterministic by construction. That is, a program in which put and get operations on LVars are the only side effects will have the same observable result every time it is run, regardless of parallel
1. INTRODUCTION

execution and schedule nondeterminism. 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 from the least-upper-bound put operation to a set of arbitrary update operations that are not necessarily idempotent (but are still inflationary and commutative), and we can generalize the get operation to allow a more general form of threshold reads. Generalizing from put to arbitrary inflationary and commutative updates turns out to be a particularly useful extension to the LVars model; I formally extend the model to support these update operations in Chapter 3, and in Chapter 4 I discuss how arbitrary update operations are useful in practice.

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 add the ability to attach *event handlers* to an LVar. When an event handler has been registered with an LVar, it causes 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.

Second, 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 writes and *freeze* operations are scheduled. However, such programs satisfy *quasi-determinism*: all executions that produce a final value (instead of raising an exception) produce the same final value.

Writes to an LVar could cause more events to occur after a group of handlers associated with that LVar has quiesced, and those events could trigger more invocations of callback functions. However, since the contents of the LVar 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 to an LVar have completed. Hence freezing and handlers make it possible to implement fixpoint computations like the graph traversal described above. Moreover, if we can ensure that a freeze does indeed happen after all writes to the LVar in question have completed, then we can ensure that the entire 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 4.2.5).
1. INTRODUCTION

1.4. The LVish library

To demonstrate the practicality of the LVars programming model, in Chapter 4 I will describe LVish, a Haskell library\(^1\) for deterministic and quasi-deterministic programming with LVars.

LVish provides a Par monad for encapsulating parallel computations.\(^2\) A Par computation can create lightweight, library-level threads that are dynamically scheduled by a custom work-stealing scheduler. LVar operations 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 write, ruling out the possibility of a run-time write-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, arrays) that support concurrent insertion, but not deletion, during Par computations. Users may also 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 LVar implementors and give examples of applications that use user-defined LVars as well as those that the library provides.

Finally, 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 results of the first phase. Using LVish, these two phases can be pipelined; moreover, the original

---

\(^1\)Available at [http://hackage.haskell.org/package/lvish](http://hackage.haskell.org/package/lvish).

\(^2\)The Par monad exposed by LVish generalizes the original Par monad exposed by the monad-par library ([http://hackage.haskell.org/package/monad-par](http://hackage.haskell.org/package/monad-par), described by Marlow et al. [39]), which allows determinism-preserving communication between threads, but only through IVars, rather than LVars.
graph exploration phase can be internally parallelized. I contrast our LVish implementation with the original sequential implementation from which it was ported and give performance results.

- Second, I describe using LVish to parallelize *PhyBin* [45], a bioinformatics application for comparing sets of phylogenetic trees that relies heavily on a parallel tree-edit distance algorithm [57]. In addition to handlers and freezing, the PhyBin application crucially relies on the aforementioned ability to perform arbitrary inflationary and commutative (but not idempotent) updates on LVars (in contrast to the idempotent put operation). We show that the performance of the parallelized PhyBin application compares favorably to existing widely-used software packages for analyzing collections of phylogenetic trees.

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) [54] for enforcing *eventual consistency* [61] of replicated objects in a distributed system. In particular, *state-based* or *convergent* replicated data types, abbreviated as CvRDTs [54, 53], leverage the mathematical properties of lattices 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. Threshold queries
1. INTRODUCTION

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 unifying abstraction for deterministic and quasi-deterministic parallel and distributed programming.

LK: Changed from “foundation” to “unifying abstraction” – otherwise it’s the same as in my proposal.

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 \( \lambda_{LVars} \), a call-by-value parallel calculus with a store of LVars that support least-upper-bound \( \text{put} \) and threshold \( \text{get} \) operations. In Chapter 3, I extend \( \lambda_{LVars} \) to add support for arbitrary update operations, event handlers, and the \text{freeze} operation, calling the resulting language \( \lambda_{LVish} \). Appendix B contains runnable versions of \( \lambda_{LVars} \) and \( \lambda_{LVish} \) implemented using the PLT Redex semantics engineering system [21] 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 \( \text{put} \) and \( \text{get} \) 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 \( \lambda_{LVars} \) language with \( \text{put} \) and \( \text{get} \).

- **quasi-deterministic**: In Chapter 3, I define quasi-determinism and give a proof of quasi-determinism for \( \lambda_{LVish} \), which adds arbitrary update operations, the \text{freeze} operation, and event handlers to the \( \lambda_{LVars} \) language of Chapter 2.
practical and parallel: In Chapter 4, I describe the interface and implementation of 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 on parallel hardware. LK: I threw in “parallel” here because I wasn’t counting it under any of the other bullets. Parallelism is a resource; we have a “parallel language semantics” in the sense that it’s a semantics that gives the scheduler lots of purchase to schedule it onto parallel hardware, and the only proof of that is in the pudding. So I’m not sure where else to support the claim of parallelism but here.

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

The material in this dissertation is based in large part on research done jointly with several collaborators, some of which appears in the following previously published papers:

LK: These are formatted in “ACM Ref” style.

- Lindsey Kuper and Ryan R. Newton. 2014. Joining forces: toward a unified account of LVars and convergent replicated data types. In the 5th Workshop on Determinism and Correctness in Parallel Programming (WoDet ’14).
1. INTRODUCTION


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. In particular, Chapter 2 is a hybrid of the first and second papers listed above, except for material in Section 2.6, which is new; Chapter 3 is based on the second paper listed above; except for everything pertaining to update operations, which is new; Chapter 4 is based on the second and fourth papers listed above; and Chapter 5 is based very loosely on the third paper listed above, but is largely new (and we have an as-yet-unpublished new paper about it). If this is actually useful information, I’ll add it; if not, I’ll leave it out.
LVars: lattice-based data structures for deterministic parallelism

Programs written using a \textit{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 (\textit{e.g.}, Cilk \cite{Cilk}) 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 \textit{futures}. For example, Haskell programs using futures by means of the \texttt{par} and \texttt{pseq} combinators can provide real speedups on practical programs while guaranteeing determinism \cite{Haskell}.\footnote{When programming with \texttt{par} and \texttt{pseq}, a language-level determinism guarantee obtains if user programs are written in the \textit{Safe Haskell} \cite{SafeHaskell} subset of Haskell (which is implemented in GHC Haskell by means of the \texttt{SafeHaskell} language pragma), and if they do not use the \texttt{IO} monad.}

Yet pure programming with futures is not ideal for all problems. Consider a \textit{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 \textit{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 \textit{et al.} \cite{Marlow}).

Since pure programming with futures is a poor fit for producer/consumer computations, one might then turn to \textit{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 \cite{DeterministicParallelJava}, for instance,
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.\footnote{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.}

In Kahn process networks (KPNs) \cite{32} and other data-flow parallel models—which are the basis for deterministic stream-processing languages such as StreamIt \cite{27}—communication among processes takes place over blocking FIFO queues with ever-increasing channel histories. Meanwhile, in single-assignment \cite{60} or IVar-based \cite{3} programming models, such as the Intel Concurrent Collections system (CnC) \cite{11} and the monad-par Haskell library \cite{39}, a shared data store of blocking single-assignment memory locations grows monotonically. Hence both programming models rely on monotonic data structures: data structures to which information can only be added and never removed, and for which the timing of updates is not observable.

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 a given lattice.\footnote{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.} This lattice determines 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 small code examples. Then, in Sections 2.3 and 2.4 I formally define $\lambda_{\text{LVar}}$, a deterministic parallel calculus with shared state, based on the call-by-value $\lambda$-calculus. The $\lambda_{\text{LVar}}$ 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 $\lambda_{\text{LVar}}$ with a lattice with one “empty” state and multiple “full” states (where $\forall i. \text{empty} < \text{full}$) results in a parallel single-assignment language. Different instantiations of the lattice result in a family of deterministic parallel languages.

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

The main technical result of this chapter is a proof of determinism for $\lambda_{\text{LVar}}$ (Section 2.5). The key lemma, Independence (Section 2.5.5), gives a kind of frame property that captures the commutative effects of LVar computations. Such a property would not hold in a typical language with shared mutable state, but holds in the setting of $\lambda_{\text{LVar}}$ because of the semantics of put and get.

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 results to shared mutable data structures. Hindley-Milner type
inference is one example: in a parallel type-inference algorithm, each type variable monotonically ac-
quires information through unification. 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 user’s 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 models. 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. Neither IVars nor streams provide a way to orchestrate this traversal. 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 us consider whether a purely functional (and therefore deterministic by construction) program can meet the specification.

Listing 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
2. LVARS: LATTICE-BASED DATA STRUCTURES FOR DETERMINISTIC PARALLELISM

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

-- Traverse each level of the graph `g' 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 == empty
    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'

-- Find the connected component containing the vertex `profile0', and
-- map `analyze` over all nodes in it:
connected_component = bf_traverse profiles empty (singleton profile0)
result = parMap analyze connected_component
```

Listing 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.

vertex are discovered—and set-unioned into the connected component—before nodes of distance two
are considered. Level-synchronization necessarily sacrifices some parallelism: parallel tasks cannot
continue discovering nodes in the component before synchronizing with all other tasks at a given dis-
tance from the start. Nevertheless, level-synchronization is a popular strategy for implementing paral-
lel breadth-first graph traversal. (In fact, the aforementioned implementation from the Parallel Boost
Graph Library [1] uses level-synchronization.)

Unfortunately, the code given in Listing 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 be-
come available to `analyze` until component discovery is finished, limiting parallelism: the first nodes
in the connected component to be discovered must go un-analyzed until all of the nodes in the con-
nected component have been traversed. 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.\textsuperscript{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 problem of separating the scheduling of producers and consumers that pure programming with futures presents [39].

\subsection*{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 will formally introduce LVars and the \textit{LVar} language and give a proof of determinism for \textit{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.

\textsuperscript{4}As intuition for this idea, consider that purely functional set data structures, such as Haskell’s \textit{Data.Set}, are typically represented with balanced trees. Unlike with lists, the structure of the tree is not known until all elements are present.
2. LVARS: LATTICE-BASED DATA STRUCTURES FOR DETERMINISTIC PARALLELISM

2.2. LVars by example

LVars [3, 46, 39] are a well-known mechanism for deterministic parallel programming.\(^5\) An IVar is a single-assignment variable \([60]\) with a blocking read semantics: an attempt to read an empty IVar will block until the IVar has been filled with a value. The “I” in IVar stands for “immutable”: an IVar can only be written to once, and thereafter its contents cannot change. LVars are a generalization of IVars: an LVar allows multiple writes, but only so long as those writes are monotonically increasing with respect to a given lattice of states.

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

\[
\text{LK: The negative vspace on these is hopefully a temporary hack.}
\]

\[
\text{let par }_\_ = \text{put } lv 3
\]

(Example 2.1)

\[
_\_ = \text{put } lv 2
\]

in get \(lv\)

Here, \text{put} and \text{get} write and read LVars, respectively, and the expression

\[
\text{let par } x_1 = e_1; x_2 = e_2; \ldots \text{ in } e_3
\]

has fork-join semantics: it launches concurrent subcomputations \(e_1, e_2, \ldots\) whose executions arbitrarily interleave, but must all complete before the expression \(e_3\) runs.

The \text{put} operation is defined in terms of the specified lattice of LVar states; it updates the LVar to the least upper bound (lub) of its current state and the new state being written. If \(lv\)'s lattice is the \(\leq\) ordering on positive integers, as shown in Figure 2.1(a), then \(lv\)'s state will always be \(\text{max}(3, 2) = 3\) by the time \text{get} \(lv\) runs, since the lub of two positive integers \(n_1\) and \(n_2\) is \(\text{max}(n_1, n_2)\). Therefore Example 2.1 will deterministically evaluate to 3, regardless of the order in which the two \text{put} operations occur.

\(^5\)IVars are so named because they are a special case of I-structures [3]—namely, those with only one cell.
Figure 2.1. 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 \texttt{getSnd} causes it to unblock and return a result.

On the other hand, if \( \text{lv}'s lattice is that shown in Figure 2.1(b), in which the lub of any two distinct positive integers is \( \top \), then Example 2.1 will deterministically raise an exception, indicating that conflicting writes to \( \text{lv} \) 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, \texttt{put lv 3} and \texttt{put lv 3}) will not raise an exception, because the lub 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 guarantee that programs behave deterministically. Consider again the lattice of Figure 2.1(a) for \( \text{lv} \), but suppose we change Example 2.1 to allow a direct, non-blocking read of the LVar to be interleaved with the two puts:
2. LVARS: LATTICE-BASED DATA STRUCTURES FOR DETERMINISTIC PARALLELISM

\[
\begin{align*}
\text{let } & \text{par }_\_ = \text{put lv} 3 \\
& \_ = \text{put lv} 2 \\
& \text{x} = \text{read lv} \quad \text{-- Not allowed!} \\
\text{in } & \text{par }_\_
\end{align*}
\]  

(Example 2.2)

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

We can then change Example 2.2 to behave deterministically using \text{get} with a threshold set:

\[
\begin{align*}
\text{let } & \text{par }_\_ = \text{put lv} 3 \\
& \_ = \text{put lv} 2 \\
& \text{x} = \text{get lv} \{3\} \\
\text{in } & \text{par }_\_
\end{align*}
\]  

(Example 2.3)

Here we chose the singleton set \{3\} as the threshold set. Since \text{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 \text{put} and \text{get}, we can arbitrarily share them between threads without introducing nondeterminism. That is, the \text{put} and \text{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 Example 2.3 is deterministic, the style of programming it illustrates 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 2.1(c).

We can then define getFst and getSnd operations for reading from the first and second entries of \( lv \):

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

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

This allows us to write programs like the following:

\[
\text{let par } \_ = \text{put } lv (\bot, 4) \\
_ = \text{put } lv (3, \bot) \\
x = \text{getSnd } lv \\
\text{in } x
\]

(Example 2.4)

In the call \( \text{getSnd } lv \), the threshold set is \( \{(\bot, 0), (\bot, 1), \ldots\} \), an infinite set. There is no risk of non-determinism because the elements of the threshold set are pairwise incompatible with respect to \( lv \)'s lattice: informally, since the second entry of \( lv \) can only be written once, no more than one state from the set \( \{(\bot, 0), (\bot, 1), \ldots\} \) can ever be reached. (I formalize this incompatibility requirement in Section 2.3.3.)

In the case of Example 2.4, \( \text{getSnd } lv \) 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 core of the LVish library (which I will discuss in Chapter 4) provides unsafe operations to the authors of LVar data structure libraries, who can then export operations like `getFst` and `getSnd` 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 lub 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 $\lambda_{\text{LVar}}$, a parallel call-by-value $\lambda$-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 $\lambda_{\text{LVar}}$ 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 $\lambda_{\text{LVar}}$, stores contain LVars. 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 $\lambda_{\text{LVar}}$ and explain how the semantics of `put` and `get` together enforce determinism in $\lambda_{\text{LVar}}$ programs.
2.3.1. Lattices. The definition of \( \lambda_{\text{LVar}} \) is parameterized by \( D \): to write concrete \( \lambda_{\text{LVar}} \) programs, we must specify the set of LVars that we are interested in working with, and an ordering on those states. Therefore \( \lambda_{\text{LVar}} \) 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 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 element. However, I use \( D \) as a shorthand for the entire 4-tuple \( (D, \sqsubseteq, \bot, \top) \) when its meaning is clear from the context.

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, \( \sqcup \) could allow for unification of partially-initialized structures.

LK: Since I’ve already talked about 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 \( D \) is one that represents the states that a single-assignment variable (that is, an IVar) can take on. The states of a natural-number-valued IVar, for instance, are the elements of the lattice in Figure 2.1(b), that is,
$D = \{ \top, \bot \} \cup \mathbb{N}, \sqsubseteq, \sqsubseteq, \top$, 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 2.1(a), on the other hand, the $\top$ state is unreachable, because the lub 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 $\lambda_{\text{LVar}}$ program, a store $S$ keeps track of the states of LVars. Each LVar is represented by a binding that maps a location $l$, drawn from a countable set $\text{Loc}$, 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 $D$.

Definition 2.1 (store, $\lambda_{\text{LVar}}$). A store is either a finite partial mapping $S : \text{Loc} \to \{ D - \{ \top \} \}$, or the distinguished element $\top$. I use the notation $S[l \mapsto d]$ to denote extending $S$ with a binding from $l$ to $d$. If $l \in \text{dom}(S)$, 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 set of states that a store can take on forms a lattice, just as $D$ does, with the empty store $\bot$ as its least element and $\top$ as its greatest element. It is straightforward to lift the $\sqsubseteq$ and $\sqcup$ operations defined on elements of $D$ to the level of stores:

In practice, different LVars in a program might correspond to different lattices (and, in the L Vish 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.
Definition 2.2 (store ordering, \(\lambda_{\text{Var}}\)). A store \(S\) is less than or equal to a store \(S'\) (written \(S \sqsubseteq S'\)) iff:

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

Definition 2.3 (lub of stores, \(\lambda_{\text{Var}}\)). The lub of stores \(S_1\) and \(S_2\) (written \(S_1 \sqcup S_2\)) is defined as follows:

- \(S_1 \sqcup S_2 = \top_S\) iff there exists some \(l \in \text{dom}(S_1) \cap \text{dom}(S_2)\) such that \(S_1(l) \sqcup S_2(l) = \top\).
- Otherwise, \(S_1 \sqcup S_2\) is the store \(S\) such that:
  - \(\text{dom}(S) = \text{dom}(S_1) \cup \text{dom}(S_2)\), and
  - For all \(l \in \text{dom}(S)\):
    \[
    S(l) = \begin{cases} 
    S_1(l) \sqcup S_2(l) & \text{if } l \in \text{dom}(S_1) \cap \text{dom}(S_2) \\
    S_1(l) & \text{if } l \notin \text{dom}(S_2) \\
    S_2(l) & \text{if } l \notin \text{dom}(S_1).
    \end{cases}
    \]

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, \(\lambda_{\text{Var}}\)). Two stores \(S\) and \(S'\) are equal iff:

1. \(S = \top_S\) and \(S' = \top_S\), or
2. \(\text{dom}(S) = \text{dom}(S')\) and for all \(l \in \text{dom}(S)\), \(S(l) = S'(l)\).

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 \(\lambda_{\text{Var}}\) program, because (as I will show in Section 2.4) an attempted write that would take the state of some location \(l\) to \(\top\) would raise an error before the write can occur.

2.3.3. Communication primitives. The \texttt{new}, \texttt{put}, and \texttt{get} operations create, write to, and read from LVars, respectively. The interface is similar to that presented by mutable references:
new extends the store with a binding for a new LVar whose initial state is ⊥, and returns the location l of that LVar (i.e., a pointer to the LVar).

put takes a pointer to an LVar and a new state and updates the LVar’s state to the lub 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 ⊤ 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 ⊤. If the LVar’s state d₁ in the lattice is at or above some d₂ ∈ T, the get operation unblocks and returns d₂. Note that d₂ is a unique element of T, for if there is another d₂' ≠ d₂ in the threshold set such that d₂' ⊆ d₁, it would follow that d₂ ∪ d₂' ⊆ d₁, and so d₂ ∪ d₂' cannot be ⊤, 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 2.1(c) shows what the “tripwire” looks like for an example get operation. The threshold set \{ (⊥, 0), (⊥, 1), \} (or a subset thereof) would pass the incompatibility test, as would the threshold set \{ (0, ⊥), (1, ⊥), \} (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 allow more programs to be expressed. 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 ⊥) is created, or a previously ⊥ 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:

Definition 2.5 (store ordering, Featherweight CnC). A store $S$ is less than or equal to a store $S'$ (written $S \sqsubseteq S'$) iff $\text{dom}(S) \subseteq \text{dom}(S')$ and for all $l \in \text{dom}(S)$, $S(l) = S'(l)$.

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 given lattice $\sqsubseteq$. In $\lambda_{\text{LVar}}$, stores may grow by updating existing bindings via repeated puts, so Definition 2.5 would be too strong; for instance, if $\bot \sqsubseteq d_1 \sqsubseteq d_2$ 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 $\lambda_{\text{LVar}}$ an LVar could take on the state $d_1$, and then later the state $d_2$, which would not be possible in Featherweight CnC.

I establish in Section 2.5 that $\lambda_{\text{LVar}}$ remains deterministic despite the relatively weak $\sqsubseteq_S$ relation given in Definition 2.2. The key to maintaining determinism is the blocking semantics of the get operation and the fact that it allows only limited observations of the state of an LVar.

2.4. $\lambda_{\text{LVar}}$: syntax and semantics

The syntax of $\lambda_{\text{LVar}}$ appears in Figure 2.2, and Figures 2.3 and 2.4 together give the operational semantics. Both the syntax and semantics are parameterized by the lattice $(D, \sqsubseteq, \bot, \top)$.

---

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

DRAFT: March 14, 2015
2. LVARS: LATTICE-BASED DATA STRUCTURES FOR DETERMINISTIC PARALLELISM

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

configurations \(\sigma ::= \langle S; e \rangle \mid \text{error}\)

expressions \(e ::= x \mid v \mid e \cdot e \mid \text{get } e \cdot e \mid \text{put } e \cdot e \mid \text{new}\)

values \(v ::= () \mid d \mid l \mid T \mid \lambda x. e\)

threshold sets \(T ::= \{d_1, d_2, \ldots\}\)

stores \(S ::= [l_1 \mapsto d_1, \ldots, l_n \mapsto d_n] \mid T_S\)

evaluation contexts \(E ::= [\] \mid E \cdot e \mid e \cdot E \mid \text{get } E \cdot e \mid \text{get } e \cdot E \mid \text{put } E \cdot e \mid \text{put } e \cdot E\)

Figure 2.2. Syntax for \(\lambda_{\text{LVars}}\).

A configuration \(\langle S; e \rangle\) comprises a store and an expression. The error configuration, written \text{error}, is a unique element added to the set of configurations, but \(\langle T_S; e \rangle\) is equal to \text{error} for all expressions \(e\). The metavariable \(\sigma\) ranges over configurations.

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 \(\lambda\)-expressions \(\lambda x. e\). A threshold set is a set \(\{d_1, d_2, \ldots\}\) of one or more elements of the specified lattice.

The operational semantics is split into two parts: a reduction semantics, shown in Figure 2.3, and a context semantics, shown in Figure 2.4.

The reduction relation \(\rightsquigarrow\) is defined on configurations. There are five rules in the reduction semantics: the E-Beta rule is standard \(\beta\)-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 \(l \notin \text{dom}(S)\) 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 \text{error}.
Given a lattice \((D, \sqsubseteq, \bot, \top)\) with elements \(d \in D\):

\[
\text{incomp}(T) \triangleq \forall d_1, d_2 \in T. (d_1 \neq d_2 \iff d_1 \sqcup d_2 = \top)
\]

\[
\sigma \mapsto \sigma'
\]

\[
\begin{array}{ll}
\text{E-Beta} & \text{E-New} \\
\langle S; (\lambda x. e) \, v \rangle \mapsto \langle S; e[x := v]\rangle & \langle S; \text{new} \rangle \mapsto \langle S[l \mapsto \bot]; \, l \rangle \\
\end{array}
\]

\[
\begin{array}{ll}
\text{E-Put} & \text{E-Put-Err} \\
S(l) = d_1 \quad d_1 \sqcup d_2 \neq \top & S(l) = d_1 \quad d_1 \sqcup d_2 = \top \\
\langle S; \text{put } l \, d_2 \rangle \mapsto \langle S[l \mapsto d_1 \sqcup d_2]; \, () \rangle & \langle S; \text{put } l \, d_2 \rangle \mapsto \text{error} \\
\end{array}
\]

\[
\begin{array}{ll}
\text{E-Get} & \\
S(l) = d_1 \quad \text{incomp}(T) \quad d_2 \in T \quad d_2 \sqsubseteq d_1 & \\
\langle S; \text{get } l \, T \rangle \mapsto \langle S; d_2 \rangle \\
\end{array}
\]

\[
\sigma \mapsto \sigma'
\]

\[
\begin{array}{ll}
\text{E-Eval-Ctxt} & \\
\langle S; \, e \rangle \mapsto \langle S'; \, e' \rangle \\
\langle S; \, E[e] \rangle \mapsto \langle S'; \, E[e'] \rangle \\
\end{array}
\]

\[
\sigma \mapsto \sigma'
\]

Figure 2.3. Reduction semantics for \(\lambda_{LVar}\).

Figure 2.4. Context semantics for \(\lambda_{LVar}\).

The incompatibility of the threshold set argument to \text{get} is enforced in the E-Get rule by the \text{incomp}(T) premise, which requires that the lub of any two distinct elements in \(T\) must be \(\top\).\(^\text{8}\)

\(^\text{8}\)Although \text{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
The context relation \( \rightarrow_{\text{context}} \) 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 \( \lambda_{\text{LVar}} \), 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 \( \lambda_{\text{LVar}} \), the arguments to \( \text{get} \), \( \text{put} \), and application expressions are implicitly evaluated concurrently.

### 2.4.1. Fork-join parallelism.

\( \lambda_{\text{LVar}} \) has a call-by-value semantics: arguments must be fully evaluated before function application (\( \beta \)-reduction, via the E-Beta rule) can occur. We can exploit this property to define the syntactic sugar \( \text{let par} \) for parallel composition that we first saw earlier in Example 2.1. With \( \text{let par} \), we can evaluate two subexpressions \( e_1 \) and \( e_2 \) in parallel before evaluating a third subexpression \( e_3 \):

\[
\text{let par } x_1 = e_1; x_2 = e_2 \text{ in } e_3 \triangleq ((\lambda x_1. (\lambda x_2. e_3)) e_1) e_2
\]

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 \( \beta \)-reduction until the operand is fully evaluated, and because it further disallows reduction under \( \lambda \)-terms (sometimes called “full \( \beta \)-reduction”). In the terminology of parallel programming, a \( \text{let par} \) expression executes both a \textit{fork} and a \textit{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 [23].

---

\(^9\)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.
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 2.5(a). The \(\lambda\)-Var 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 2.5(b). This adds the ability to construct arbitrary non-series-parallel dependency graphs, just as with *first-class futures* [55].

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

**2.4.2. 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 \(lv\):
2. LVARS: LATTICE-BASED DATA STRUCTURES FOR DETERMINISTIC PARALLELISM

\[
\text{let } \text{par } = \text{put } lv \ d_1 \\
\quad = \text{put } lv \ d_2 \\
\quad x = \text{get } lv \ \{d_1, d_2\} \\
\text{in } x
\]

(Example 2.5)

Eventually, Example 2.5 is guaranteed to raise \textbf{error} by way of the E-Put-Err rule, because \(d_1 \sqcup d_2 = \top\). Before that happens, though, \(\text{get } lv \ \{d_1, d_2\}\) could return either \(d_1\) or \(d_2\). Therefore, \(\text{get}\) can behave nondeterministically— but this behavior is not observable in the final outcome of the program, since one of the two \(\text{puts}\) will raise \textbf{error} before the \(x\) in the body of the \text{let } \text{par} can be evaluated, and under our definition of observable determinism, only the final outcome of a program counts.

2.5. Proof of determinism for \(\lambda_{LVars}\)

The main technical result of this chapter is a proof of determinism for the \(\lambda_{LVars}\) language. The determinism theorem says that if two executions starting from a given configuration \(\sigma\) terminate in configurations \(\sigma'\) and \(\sigma''\), then \(\sigma'\) and \(\sigma''\) 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 \(\lambda_{LVars}\), 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 \(\lambda_{LVars}\), due to the generality of LVars and other differences between the \(\lambda_{LVars}\) 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 \( l \in \text{Loc} \) 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 reduc-
tion semantics is nondeterministic with respect to which locations are allocated. Since this is not a kind
of nondeterminism that we care about, we work modulo an arbitrary permutation on locations.

Recall from Section 2.3.2 that we have a countable set of locations \( \text{Loc} \). Then, a permutation is defined
as follows:

Definition 2.6 (permutation, \( \lambda_{\text{LVar}} \)). A permutation is a function \( \pi : \text{Loc} \rightarrow \text{Loc} \) such that:

1. \( \pi \) is invertible, that is, there is an inverse function \( \pi^{-1} : \text{Loc} \rightarrow \text{Loc} \) with the property that \( \pi(l) = l' \)
   \[ \text{iff } \pi^{-1}(l') = l \]; and
2. \( \pi \) is the identity on all but finitely many elements of \( \text{Loc} \).

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 \( \pi \) is a bijection from \( \text{Loc} \) to \( \text{Loc} \) such that it is the identity on all but finitely many elements.

Definitions 2.7, 2.8, and 2.9 lift Definition 2.6 to expressions, stores, and configurations, respectively.
There is nothing surprising about these definitions: to apply a permutation \( \pi \) to an expression, we
just apply \( \pi \) to any locations that occur in the expression. We can also lift \( \pi \) to evaluation contexts,
structurally: \( \pi([]) = [] \), \( \pi(E \ e = \pi(E) \ \pi(e)) \), and so on. To lift \( \pi \) to stores, we apply \( \pi \) to all locations
in the domain of the store. (We do not have to do any renaming in the codomain of the store, since
location names cannot occur in elements of the lattice \( D \) and hence cannot occur in the contents of
other store locations.) Since \( \pi \) is a bijection, it follows that if some location \( l \) is not in the domain of
some store $S$, then $\pi(l) \notin \text{dom}((\pi(S)))$, LK: Do I need to spell out why this is true? a fact that will be useful to us shortly.

Definition 2.7 (permutation of an expression, $\lambda_{\text{LVar}}$). A permutation of an expression $e$ is a function $\pi$ defined as follows:

$$
\begin{align*}
\pi(x) & \triangleq x \\
\pi(\ ) & \triangleq () \\
\pi(d) & \triangleq d \\
\pi(l) & \triangleq \pi(l) \\
\pi(T) & \triangleq T \\
\pi(\lambda x. e) & \triangleq \lambda x. \pi(e) \\
\pi(e_1 e_2) & \triangleq \pi(e_1) \pi(e_2) \\
\pi(\text{get } e_1 e_2) & \triangleq \text{get } \pi(e_1) \pi(e_2) \\
\pi(\text{put } e_1 e_2) & \triangleq \text{put } \pi(e_1) \pi(e_2) \\
\pi(\text{new}) & \triangleq \text{new}
\end{align*}
$$

LK: It’s fine to just say that $\pi(\text{new})$ 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 of a store, $\lambda_{\text{LVar}}$). A permutation of a store $S$ is a function $\pi$ defined as follows:

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

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

DRAFT: March 14, 2015
With these definitions in place, I can prove Lemma 2.1, which says that the names of locations in a configuration do not affect whether or not that configuration can take a step: a configuration $\sigma$ can step to $\sigma'$ exactly when $\pi(\sigma)$ can step to $\pi(\sigma')$.

Lemma 2.1 (Permutability, $\lambda_{LVar}$). For any finite permutation $\pi$,

1. $\sigma \rightarrow \sigma'$ if and only if $\pi(\sigma) \rightarrow \pi(\sigma')$.
2. $\sigma \leftarrow \sigma'$ if and only if $\pi(\sigma) \leftarrow \pi(\sigma')$.

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

Because the names of locations in a configuration do not affect whether it can step, we can rename locations as needed, which will be important later on when proving the confluence lemmas of Section 2.5.8.10

2.5.2. Internal determinism. My goal is to show that $\lambda_{LVar}$ is deterministic according to the definition of observable determinism that I gave in Chapter 1—that is, that a $\lambda_{LVar}$ program always evaluates to the same value. In the context of $\lambda_{LVar}$, 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 $\lambda_{LVar}$ value, or because it is a “stuck” configuration that cannot step because no rule of the operational semantics applies. In $\lambda_{LVar}$, 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. Borrowing terminology from Blelloch et

---

10For another example of using permutations in the metatheory of a language to account for an allocator’s nondeterministic choice of locations in an otherwise deterministic setting, see Krishnaswami [34].
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.\(^{11}\) Although \(\lambda_{LVar}\) is not internally deterministic, all of its internal non-determinism 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 \(\lambda_{LVar}\) expression (a \(\text{let par}\) expression, for instance) to be evaluated in parallel.

But, leaving aside evaluation contexts for the moment—we will return to them in the following section—let us focus on the rules of the reduction semantics in Figure 2.3. Here we can see that if a given configuration can step by the reduction semantics, then there is only one rule by which it can step, and only one configuration to which it can step. The only 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, which we will later use in the proof of Strong Local Confluence (Lemma 2.8).

Lemma 2.2 (Internal Determinism, \(\lambda_{LVar}\)). If \(\sigma \rightarrow \sigma'\) and \(\sigma \rightarrow \sigma''\), then there is a permutation \(\pi\) such that \(\sigma' = \pi(\sigma'')\).

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

### 2.5.3. Locality

In order to prove determinism for \(\lambda_{LVar}\), 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 that decomposes into redex and context in multiple ways. Suppose that we have an expression that decomposes into redex and context in multiple ways.
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: $\langle S; E_1[e_1] \rangle \rightarrow \langle S_1; E_1'[e_1] \rangle$, and $\langle S; E_2[e_2] \rangle \rightarrow \langle S_2; E_2'[e_2] \rangle$.

The key observation we can make here is that the $\rightarrow$ 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, $\lambda_{\text{Var}}$). If $\langle S; E_1[e_1] \rangle \rightarrow \langle S_1; E_1'[e_1] \rangle$ and $\langle S; E_2[e_2] \rangle \rightarrow \langle S_2; E_2'[e_2] \rangle$ and $E_1[e_1] = E_2[e_2]$, where $E_1 \neq E_2$, then there exist evaluation contexts $E'_1$ and $E'_2$ such that:

- $E'_1[e_1] = E_2[e_2]$
- $E'_2[e_2] = E_1[e_1]$
- $E'_1[e_1'] = E'_2[e_2']$

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 $\rightarrow$ relation, the store can only grow with respect to the $\sqsubseteq_S$ ordering.

Lemma 2.4 (Monotonicity, $\lambda_{\text{Var}}$). If $\langle S; e \rangle \rightarrow \langle S'; e' \rangle$, 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. Figure 2.6 shows a frame rule, due to O’Hearn et al. [48], which captures the idea of local reasoning about programs that alter state. In it, $C$ is a program, and $\{p\} C \{q\}$ is a Hoare triple (in the style of Hoare logic [30]) specifying the behavior of $C$: it says that if the assertion $p$ is true before $C$ runs, then the assertion $q$ will be true afterwards. For example, $p$ and $q$ might respectively describe the state of the heap before and after a heap location is updated by $C$. 

DRAFT: March 14, 2015
2. LVARS: LATTICE-BASED DATA STRUCTURES FOR DETERMINISTIC PARALLELISM

Frame rule:

\[
\frac{\{p\} C \{q\}}{\{p * r\} C \{q * r\}}
\]

Lemma 2.5 (Independence), simplified:

\[
\langle S; e \rangle \iff \langle S'; e' \rangle \\
\langle S \sqcup S''; e \rangle \iff \langle S' \sqcup S''; e' \rangle
\]

Figure 2.6. Comparison of O’Hearn et al.’s frame rule \[48\] and a simplified version of the Independence lemma. The separating conjunction connective \(*\) in the frame rule requires that its arguments be disjoint; the Independence lemma uses the \(\sqcup\) operation in place of \(*\).

Given a program \(C\) with precondition \(p\) and postcondition \(q\), the frame rule tells us that running \(C\) starting from a state satisfying the precondition \(p * r\) will result in a state satisfying the postcondition \(q * r\). These two assertions use the *separating conjunction* connective \(*\), which combines two assertions that can be satisfied in a *non-overlapping* manner. For instance, the assertion \(p * r\) is satisfied by a heap if the heap can be split into two non-overlapping parts satisfying \(p\) and \(r\), respectively.

Therefore, if \(C\) can run safely starting from a state satisfying \(p\) and end in a state satisfying \(q\), then it does not do any harm to also have the disjoint property \(r\) be true when \(C\) runs: the truth of \(r\) will not interfere with the safe execution of \(C\). Furthermore, if \(r\) is true to begin with, running \(C\) will not interfere with the truth of \(r\). The frame rule gets its name from the fact that \(r\) is a “frame” around \(C\): everything that is not explicitly changed by \(C\) is part of the frame and is inviolate.\[12\] O’Hearn et al. refer

\[12\]The "frame" terminology was originally introduced in 1969 by McCarthy and Hayes \[41\], who observed that specifying only what is changed by an action does not generally allow an intelligent agent to conclude that nothing else is changed; they called this dilemma the *frame problem.*
to the resources (such as heap locations) actually used by \( C \) as the “footprint” of \( C \); \( r \) is an assertion about resources outside of that footprint.

The Independence lemma establishes a similar “frame property” for \( \lambda_{LVars} \) 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 \( S \sqcup S'' \): we know both that \( e \) will update the store to \( S' \sqcup S'' \), and that \( e \) will step to \( e' \) as it did before. Here, \( S \sqcup S'' \) is the lub 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.\(^\text{13}\)

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 \( \langle S; e \rangle \mapsto \langle S'; e' \rangle \) iff \( (\text{dom}(S') - \text{dom}(S)) \cap \text{dom}(S'') = \emptyset \).

**Lemma 2.5 (Independence).** If \( \langle S; e \rangle \mapsto \langle S'; e' \rangle \) (where \( \langle S'; e' \rangle \neq \text{error} \)), then for all \( S'' \) such that \( S'' \) is non-conflicting with \( \langle S; e \rangle \mapsto \langle S'; e' \rangle \) and \( S' \sqcup_S S'' \neq T_S \):

\[
\langle S \sqcup_S S'' ; e \rangle \mapsto \langle S' \sqcup_S S'' ; e' \rangle.
\]

**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 \( \langle S'; e' \rangle \neq \text{error} \), we do not need to consider the E-Put-Err rule. See Section A.5. \( \square \)

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

\( ^\text{13} \)See Section 6.6 for a more detailed discussion of frame properties and where they manifest in the LVars model.
Lemma 2.6 (Clash). If \( \langle S; e \rangle \rightarrow \langle S'; e' \rangle \) (where \( \langle S'; e' \rangle \neq \text{error} \)), then for all \( S'' \) such that \( S'' \) is non-conflicting with \( \langle S; e \rangle \), \( \langle S'; e' \rangle \) and \( S' \sqcup_{S} S'' = \top_{S} \):

\[
\langle S \sqcup_{S} S'' ; e \rangle \rightarrow^{i} \text{error}, \text{ where } i \leq 1.
\]

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 \( \langle S'; e' \rangle \neq \text{error} \), 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 \( \text{error} \), then evaluating \( e \) in the context of some larger store will also result in \( \text{error} \).

Lemma 2.7 (Error Preservation, \( \lambda_{\text{Var}} \)). If \( \langle S; e \rangle \rightarrow \text{error} \) and \( S \sqsubseteq S' \), then \( \langle S'; e \rangle \rightarrow \text{error} \).

Proof. Suppose \( \langle S; e \rangle \rightarrow \text{error} \) and \( S \sqsubseteq S' \). We are required to show that \( \langle S'; e \rangle \rightarrow \text{error} \).

By inspection of the operational semantics, the only rule by which \( \langle S; e \rangle \) can step to \( \text{error} \) is E-Put-Err. Hence \( e = \text{put } l d_{2} \). From the premises of E-Put-Err, we have that \( S(l) = d_{1} \). Since \( S \sqsubseteq S' \), 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, \( \langle S'; \text{put } l d_{2} \rangle \rightarrow \text{error} \), as we were required to show.

2.5.8. Confluence. Lemma 2.8, the Strong Local Confluence lemma, says that if a configuration \( \sigma \) can step to configurations \( \sigma_{a} \) and \( \sigma_{b} \), then there exists a configuration \( \sigma_{c} \) that \( \sigma_{a} \) and \( \sigma_{b} \) can each reach in at most one step, modulo a permutation on the locations in \( \sigma_{b} \). Lemmas 2.9 and 2.10 then generalize that result to arbitrary numbers of steps.

The structure of this part of the proof differs from the Budimlić et al. determinism proof for Featherweight CnC in two ways. First, Budimlić et al. prove a diamond property, in which \( \sigma_{a} \) and \( \sigma_{b} \) each step to \( \sigma_{c} \) 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 \( \lambda_{LVar} \); so, instead, I prove the weaker “at most one step” property directly.

Second, Budimlić et al. do not have to deal with permutations in their proof, because the Featherweight CnC language does no allocation; there is no counterpart to \( \lambda_{LVar} \)’s \texttt{new} expression in Featherweight CnC. Instead, Featherweight CnC models the store as a pre-existing array of locations, where every location has a default initial value of \( \perp \). Because there is no way (and no need) to allocate new locations in Featherweight CnC, it is never the case that two subexpressions independently happen to allocate locations with the same name—which is exactly the situation that requires us to be able to rename locations in \( \lambda_{LVar} \). In fact, that situation is what makes the entire notion of permutations described in Section 2.5.1 a necessary part of the metatheory of \( \lambda_{LVar} \).

Taking the approach of Featherweight CnC, and therefore avoiding allocation entirely, would simplify both the \( \lambda_{LVar} \) language and its determinism proof. On the other hand, when programming with the LVish Haskell library of Chapter 4, one does have to explicitly create and allocate new LVars by calling the equivalent of \texttt{new}, and so by modeling the \texttt{new} operation in \( \lambda_{LVar} \), we keep the semantics a bit more faithful to the implementation.

**Lemma 2.8 (Strong Local Confluence).** If \( \sigma \xrightarrow{a} \sigma_a \) and \( \sigma \xrightarrow{b} \sigma_b \), then there exist \( \sigma_c, i, j, \pi \) such that \( \sigma_a \xrightarrow{i} \sigma_c \) and \( \pi(\sigma_b) \xrightarrow{j} \sigma_c \) and \( i \leq 1 \) and \( j \leq 1 \).

**Proof.** Since the original configuration \( \sigma \) can step in two different ways, its expression decomposes into redex and context in two different ways: \( \sigma = \langle S; E_a[e_{a1}] \rangle = \langle S; E_b[e_{b1}] \rangle \), where \( E_a[e_{a1}] = E_b[e_{b1}] \), but \( E_a \) and \( E_b \) may differ and \( e_{a1} \) and \( e_{b1} \) may differ. In the special case where \( E_a = E_b \), the result follows by Internal Determinism (Lemma 2.2).

If \( E_a \neq E_b \), we can apply the Locality lemma (Lemma 2.3); at a high level, it shows that \( e_{a1} \) and \( e_{b1} \) can be evaluated independently within their contexts. The proof is then by a double case analysis on the rules of the reduction semantics by which \( \langle S; e_{a1} \rangle \) steps and by which \( \langle S; e_{b1} \rangle \) steps. In order to
combine the results of the two independent steps, the proof makes use of the Independence lemma (Lemma 2.5). The most interesting case is that in which both steps are by the E-New rule and they allocate locations with the same name. In that case, we can use the Permutability lemma (Lemma 2.1) to rename locations so as not to conflict. See Section A.7.

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

Proof. By induction on \( m \); see Section A.8.

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

Proof. By induction on \( n \); see Section A.9.

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

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 \( \sigma \rightarrow^* \sigma' \) and \( \sigma \rightarrow^* \sigma'' \), and neither \( \sigma' \) nor \( \sigma'' \) can take a step, then there exists \( \pi \) such that \( \sigma' = \pi(\sigma'') \).

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

By Permutability (Lemma 2.1), \( \sigma'' \) can step iff \( \pi(\sigma'') \) can step, so since \( \sigma'' \) 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 $\lambda_{\text{LVar}}$, 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 $\lambda_{\text{LVar}}$ 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 $\lambda_{\text{LVar}}$ 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 $\lambda_{\text{LVar}}$ 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 $\lambda_{\text{LVar}}$. Such a property could be paired with Theorem 2.1 to guarantee that if one run of a given $\lambda_{\text{LVar}}$ program terminates in a non-error configuration $\sigma$, then every run of that program terminates in $\sigma$. (The “non-error configuration” condition is necessary because it is possible to construct a $\lambda_{\text{LVar}}$ 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 $\lambda_{\text{LVar}}$ shows that adding LVars (with their accompanying new/put/get operations) to an existing deterministic parallel language (the $\lambda$-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 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.
2. LVARS: LATTICE-BASED DATA STRUCTURES FOR DETERMINISTIC PARALLELISM

Consider an LVar based on the lattice of Figure 2.1(a). Under the 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 family of update operations \(u_i : D \rightarrow D\), which must meet the following two conditions:

\[
\begin{align*}
&\forall d, i. \ d \sqsubseteq u_i(d) \\
&\forall d, i, j. \ u_i(u_j(d)) = u_j(u_i(d))
\end{align*}
\]

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 timing of updates must not be observable.\(^{14}\)

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.

\(^{14}\)Of course, commutativity of updates alone is not enough to assure that the timing of updates is not observable; for that we also need threshold reads.
In fact, the \texttt{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 lub. However, when generalizing LVars to support update operations, we must keep in mind that \texttt{put} operations do not necessarily mix with arbitrary update operations on the same LVar. For example, consider a family of update operations \( \{u_{(+1)}, u_{(+2)}, \ldots\} \) 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 \texttt{put} of 4 and a \( u_{(+1)} \) do not commute: if we start with an initial state of 0 and the \texttt{put} occurs first, then the state of the LVar changes to 4 since \( \text{max}(0, 4) = 4 \), 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 \( \text{max}(1, 4) = 4 \). 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 responsibility to ensure that they commute. For example, the LVish Haskell library of Chapter 4 provides a set data structure, \texttt{Data.LVar.Set}, that supports only \texttt{put}, whereas the counter data structure \texttt{Data.LVar.Counter} supports only increments; an attempt to call \texttt{put} on a \texttt{Counter} would be ruled out by the type system. However, \textit{composing} LVars that support different families of update operations is fine. For example, an LVar could represent a monotonically growing collection (which supports \texttt{put}) of counter LVars, where each counter is itself monotonically increasing and supports only increment. Indeed, the PhyBin case study that I describe in Section 4.6 uses just such a collection of counters.

In Chapter 3, in addition to extending \( \lambda_{\text{LVar}} \) to support the new features of \textit{freezing} and \textit{event handlers}, I generalize the \texttt{put} operation to allow arbitrary update operations. More precisely, I replace the \texttt{put} operation with a family of operations \texttt{put}_i, with each corresponding to an update operation \textit{u}_i. The resulting generalized language definition is therefore parameterized not only by a given lattice, but
also by a given family of update operations. Furthermore, as we will see in Section 3.3.5, we will need to generalize the Independence lemma (Lemma 2.5) in order to accommodate this change to the language.

LK: The Chapter 3 \(\lambda_{\text{LVish}}\) 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 \(\lambda_{\text{LVar}}\) language here, with \texttt{put} replaced with \texttt{put_{i}}, 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 \(\lambda_{\text{LVish}}\) comes from freezing, not from update. I hope that this is OK! (There's a footnote about this in Section 3.3.5.)

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 will 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 \((\text{Bot}, \text{Bot})\) 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, 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 \((T, \text{Bot})\) and \((\text{Bot}, F)\) is \((T, F)\), and the lub of \((T, \text{Bot})\) and \((F, \text{Bot})\) is Top since the overlapping T and F values conflict. The \texttt{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 us consider what observations it is possible to make of an AndLV under our existing definition of
Figure 2.7. The 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.

The states \((T, T)\), \((T, F)\), \((F, T)\), and \((F, F)\) are all pairwise incompatible with one another, and so \(\{(T, T), (T, F), (F, T), (F, F)\}\)—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, \((F, \text{Bot})\) or \((\text{Bot}, F)\) was written, since no later write could change the fact that the result of the whole computation would be “false”.\(^{15}\) Unfortunately, we cannot include \((F, \text{Bot})\) or \((\text{Bot}, F)\) in our threshold set, because 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

\(^{15}\)Actually, this is not quite true: a write of \((F, \text{Bot})\) followed by a write of \((T, \text{Bot})\) 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 \((T, \text{Bot})\) 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 2.4.2).
states that the data structure might be in when a state containing at least one F value has been written—that is, the set \{ (F, Bot), (Bot, F), (F, T), (T, F), (F, 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 \{ (T, T) \}, since we have to wait until we reach the state (T, T) to know that the result is “true”; a state like (T, 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:

\[
\{ \{ (F, Bot), (Bot, F), (F, T), (T, F), (F, F) \} \cup \{ (T, T) \} \}
\]

We redefine the semantics of \texttt{get} as follows: if an LVar’s state reaches (or surpasses) any state or states in a particular activation set in the threshold set, \texttt{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 \texttt{get} operation will block. In the case of our AndLV, as soon as either input writes a state containing an F, our \texttt{get} will unblock and return the first activation set, that is, \{ (F, Bot), (Bot, F), (F, T), (T, F), (F, 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 (T, Bot) and (Bot, T), the \texttt{get} will unblock and return \{ (T, T) \}.

In practice, the value returned from the \texttt{get} could be more meaningful to the client—for instance, a Haskell implementation could return \texttt{False} instead of returning the actual activation set that corresponds to “false”. However, the translation from \{ (F, Bot), (Bot, F), (F, T), (T, F), (F, F) \} to \texttt{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, (T, F), (F, 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 $Q = \{(T, T)\}$ and $R = \{(F, Bot), (Bot, F), (F, T), (T, F), (F, F)\}$, the lub of $(T, T)$ and $r$ must be Top, where $r$ is any element of $R$. We can easily verify that this is the case.

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:

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

A get corresponding to this so-called threshold set will unblock and return $\{(F, Bot), (Bot, F)\}$ as soon as a state containing an $F$ is reached, and $\{(T, Bot), (Bot, T)\}$ as soon as a state containing a $T$ is reached. If, for instance, the left input writes $(F, Bot)$ and the right input writes $(Bot, T)$, and these writes occur in arbitrary order, the threshold read will return a nondeterministic result, depending on the order of the two writes. But if get uses the properly pairwise-incompatible threshold set that has $Q$ and $R$ as its two activation sets, it will block until the write of $(F, Bot)$ arrives, and then will deterministically return $Q$, the “false” activation set, regardless of whether the write of $(Bot, T)$ 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

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

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 makes 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 $f_i : D \rightarrow C$ (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 $l \{f_1, f_2, \ldots \}$, 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 information can only be added, never removed, and the timing with which information is added (and hence the order in which it 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 lub 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.

We want to use LVars to implement fixpoint computations like the parallel graph traversal of Section 2.1. But we cannot do so using only least-upper-bound writes and threshold reads, because in order to determine when the set of traversed nodes in the graph has reached a fixpoint, we need to be able to see the exact contents of that set, and it is impossible to learn the exact contents of the set using only threshold reads.

In this chapter, I describe two extensions to the basic LVars model of Chapter 2 that give us a new way to approach problems like the parallel graph traversal problem. First, I add the ability to attach event
handlers to an LVar that allow callback functions to run in response to updates to the LVar. We say that a group of event handlers is quiescent when no callbacks are currently enabled to run. Second, I add a new primitive operation, freeze, that returns the exact contents of an LVar without blocking. Using freeze to read an LVar comes with the following tradeoff: once an LVar has been read, it is frozen, and any further writes that would change its value instead throw an exception.

The threshold reads that we have seen so far encourage a synchronous, pull model of programming in which threads ask specific questions of an LVar, potentially blocking until the answer is “yes”. The addition of handlers, quiescence, and freezing, by contrast, enables an asynchronous, push model of programming. We will refer to this extended programming model as the LVish programming model. Because quiescence makes it possible to tell when the fixpoint of a computation has been reached, the LVish model is particularly well suited to problems like the graph traversal problem that we saw in Section 2.1.

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

¹The same is true for quiescence detection, as we will see in Section 3.1.2.
3. QUASI-DETERMINISTIC AND EVENT-DRIVEN PROGRAMMING WITH LVARS

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 \texttt{freeze} construct is only used as the very final 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 \textit{LVish model}. The rest of this chapter introduces the LVish programming model, first informally through a series of examples, and then formally, by extending the $\lambda_{LV}$ calculus of Chapter 2 to add support for handlers, quiescence, freezing, and the arbitrary update operations described in Section 2.6.1, resulting in a calculus I call $\lambda_{LVish}$. I will also return to our parallel graph traversal problem and show an solution implemented using the LVish Haskell library.

Finally, the main technical result of this chapter is a proof of quasi-determinism for $\lambda_{LVish}$. The key to the proof is a generalized version of the Independence lemma of Chapter 2 that accounts for both freezing and the arbitrary update operations that $\lambda_{LVish}$ allows.

### 3.1. \textit{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 \textit{event} for an LVar can be represented by a lattice element; the event \textit{occurs} when the LVar’s current value reaches a point at or above that lattice element. An \textit{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 2.1(a) from Chapter 2. Suppose that lv is an LVar whose states correspond to this lattice. The expression

(Example 3.6) \[ \text{addHandler } lv \{ 1, 3, 5, \ldots \} (\lambda x. \text{put } lv \ x + 1) \]

registers a handler for lv that executes the callback function \( \lambda x. \text{put } lv \ x + 1 \) for each odd number that lv is at or above. When Example 3.6 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 \( \leq 4 \), 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 \( \lambda_{LVish} \), but with the addition of addHandler):

(Example 3.7)

\[
\begin{align*}
\text{let } &\_ = \text{put } lv \ 0 \\
&\_ = \text{put } lv \ 1 \\
&\_ = \text{addHandler } lv \{ 0, 1 \} (\lambda x. \text{if } x = 0 \text{ then put } lv \ 2) \\
\text{in get } lv \ \{ 2 \}
\end{align*}
\]

Can Example 3.7 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

\(^2\)Like threshold sets, these event sets are a mathematical modeling tool only; they have no explicit existence in the LVish library implementation.
with the puts. By instead executing a handler’s callback once for each and every element in its event set below or at the LVars value, we guarantee quasi-determinism—and, for Example 3.7, 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:

\[
\text{forEach} = \lambda lv. \lambda f. \text{addHandler} \ lv \{\{0\}, \{1\}, \{2\}, \ldots \} \ f
\]

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. 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:

```plaintext
let h = newPool
    in addHandlerInPool 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 \texttt{newPool} function, and handlers are registered with \texttt{addHandlerInPool}, a variant of \texttt{addHandler} that takes a handler pool as an additional argument. Finally, \texttt{quiesce} takes a handler pool as its argument and blocks until all of the handlers in the pool have reached a quiescent state.

Whether or not a handler is quiescent is a non-monotonic property: we can move in and out of quiescence as more writes 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 writes will not arrive to move the LVar’s state upwards in the lattice and trigger more callbacks. Early quiescence poses no risk to quasi-determinism, however, because \texttt{quiesce} does not yield any information about which events have been handled—any such questions must be asked through LVar functions like \texttt{get}. In practice, \texttt{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 \texttt{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 \texttt{freeze}, which takes an LVar as its sole argument and returns the exact value of the LVar as its result. Any writes that would change the value of a frozen LVar instead raise an exception, and it is the potential for races between such writes and \texttt{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:

\[
\texttt{freezeAfter} = \lambda lv. \lambda Q. \lambda f. \\text{let } h = \texttt{newPool} \\
\text{ in } \texttt{addHandlerInPool} h \ lv \ Q \ f; \\
\texttt{quiesce} h; \\
\texttt{freeze} lv
\]

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. QUASI-DETERMINISTIC AND EVENT-DRIVEN PROGRAMMING WITH LVARS

traverse :: Graph \rightarrow NodeLabel \rightarrow Par (Set NodeLabel)
traverse g startNode = do
  seen ← newEmptySet
  insert startNode seen -- Kick things off
  let handle node = mapM (λv → insert v seen) (neighbors g node)
  freezeSetAfter seen handle

Listing 3.1. A parallel graph traversal in LVish.

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 simple fashion shown in Listing 3.1. This code, written using the LVish Haskell library, discovers (in parallel) the set of nodes in a graph $g$ reachable from a given node $\text{startNode}$, and is guaranteed to produce a deterministic result.\footnote{In the LVish library, the $\text{Par}$ type constructor is the monad in which LVar computations run; see Chapter 4 for details.} It works by creating a fresh $\text{Set LVar}$ (corresponding to a lattice whose elements are sets, with set union as lub), and seeding it with the starting node.

TODO: As it is, this is not actually running LVish code – although the versions in Section 4.2.6 are. We could point ahead to that, or we could make this running code, but one way or another we either need to acknowledge this.

The $\text{freezeSetAfter}$ function implements a set-specific variant of the freeze-after pattern described above. It combines freezing and event handlers: first, it registers the callback $\text{handle}$ as a handler for the $\text{seen}$ 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 $\text{seen}$ set has reached a fixpoint—$\text{freezeSetAfter}$ will freeze the set and return its exact value.

3.2. LVish, formally

In this section, I present $\lambda_{\text{LVish}}$, a core calculus for the LVish programming model. It extends the $\lambda_{\text{LVar}}$ language of Chapter 2. Rather than modeling the full ensemble of event handlers, handler pools, quiescence, and freezing as separate primitives in $\lambda_{\text{LVish}}$, though, 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 the arbitrary update operations of Section 2.6.1, which are inflationary and commutative but do not necessarily compute a lub.

3.2.1. **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 $\lambda_{\text{Vish}}$, an LVar’s state is a pair $(d, frz)$, where $d$ is an element of the 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 given ordering $\sqsubseteq$ on elements of $D$. Every element of $D$ is “freezable” except $\top$. Informally:

- Two unfrozen states are ordered according to the given $\sqsubseteq$; 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 $(d, \text{false})$ is less than or equal to a frozen state $(d', \text{true})$ 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 lub operation that $\sqsubseteq_p$ induces. Definitions 3.1 and 3.2 and Lemmas 3.1 and 3.2 formalize this notion.

Definition 3.1 (lattice with status bits). Suppose $(D, \sqsubseteq, \bot, \top)$ is a lattice. We define an operation $\text{Freeze}(D, \sqsubseteq, \bot, \top) \triangleq (D_p, \sqsubseteq_p, \bot_p, \top_p)$ as follows:
3. QUASI-DETERMINISTIC AND EVENT-DRIVEN PROGRAMMING WITH LVARS

(1) \( D_p \) is a set defined as follows:

\[
D_p \triangleq \{(d, \text{frz}) \mid d \in (D - \{\top\}) \land \text{frz} \in \{\text{true, false}\}\}
\cup \{(\top, \text{false})\}
\]

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

\[
\begin{align*}
(d, \text{false}) \sqsubseteq_p (d', \text{false}) & \iff d \sqsubseteq d' \\
(d, \text{true}) \sqsubseteq_p (d', \text{true}) & \iff d = d' \\
(d, \text{false}) \sqsubseteq_p (d', \text{true}) & \iff d \sqsubseteq d' \\
(d, \text{true}) \sqsubseteq_p (d', \text{false}) & \iff d' = \top
\end{align*}
\]

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

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

Lemma 3.1 (Partition of \( D_p \)). If \( (D, \sqsubseteq, \bot, \top) \) is a lattice and \( (D_p, \sqsubseteq_p, \bot_p, \top_p) = \text{Freeze}(D, \sqsubseteq, \bot, \top) \), and \( X = D - \{\top\} \), 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. \qed
Definition 3.2 (lub of states, $\lambda_{\text{LVish}}$). We define a binary operator $\sqcup_p \in D_p \times D_p \to D_p$ as follows:

$$(d_1, \text{false}) \sqcup_p (d_2, \text{false}) \triangleq (d_1 \sqcup d_2, \text{false})$$

$$(d_1, \text{true}) \sqcup_p (d_2, \text{true}) \triangleq \begin{cases} (d_1, \text{true}) & \text{if } d_1 = d_2 \\ (T, \text{false}) & \text{otherwise} \end{cases}$$

$$(d_1, \text{false}) \sqcup_p (d_2, \text{true}) \triangleq \begin{cases} (d_2, \text{true}) & \text{if } d_1 \sqsubseteq d_2 \\ (T, \text{false}) & \text{otherwise} \end{cases}$$

$$(d_1, \text{true}) \sqcup_p (d_2, \text{false}) \triangleq \begin{cases} (d_1, \text{true}) & \text{if } d_2 \sqsubseteq d_1 \\ (T, \text{false}) & \text{otherwise} \end{cases}$$

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

Lemma 3.2 (Lattice structure). If $(D, \sqsubseteq, \bot, T)$ is a lattice and $(D_p, \sqsubseteq_p, \bot_p, T_p) = \text{Freeze}(D, \sqsubseteq, \bot, T)$, then:

1. $\sqsubseteq_p$ is a partial order over $D_p$.
2. Every nonempty finite subset of $D_p$ has a lub.
3. $\bot_p$ is the least element of $D_p$.
4. $T_p$ is the greatest element of $D_p$.

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

Proof. See Section A.10.

3.2.2. Update operations. $\lambda_{\text{LVish}}$ generalizes the put operation of $\lambda_{\text{LVar}}$ to a family of operations $\text{put}_i$, in order to allow the generalized update operations of Section 2.6.1 that are commutative and inflationary, but do not necessarily compute a lub. To make this possible we parameterize $\lambda_{\text{LVish}}$ not only by the lattice $(D, \sqsubseteq, \bot, T)$, but also by a set $U$ of update operations, as discussed previously in Section 2.6.1:

Definition 3.3 (set of update operations). Given a lattice $(D, \sqsubseteq, \bot, T)$ with elements $d \in D$, a set of update operations $U$ is a set of functions $u_i : D \to D$ meeting the following conditions:
The first of the conditions in Definition 3.3 says that each update operation is inflationary with respect to \( \sqsubseteq \), and the second condition says that update operations commute with each other. Every set of update operations always implicitly contains the identity function.

If we want to recover the original semantics of \( \text{put} \), 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 \( \{ u_{(+1)}, u_{(+2)}, \ldots \} \), 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 lub writes as well as non-idempotent increments. (When a write is specifically a lub write, I will continue to use the notation \( \text{put} \), without the subscript.)

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

More specifically, since \( l \) points to a state \( (d, \text{frz}) \) instead of an element \( d \), \( \text{put}_i \ l \) must perform \( u_{p_i} \), a lifted version of \( u_i \) that applies to states. Given \( U \), we define the set \( U_p \) of lifted operations as follows:

Definition 3.4 (set of state update operations). Given a set \( U \) of update operations \( u_i \), the corresponding set of state update operations \( U_p \) is a set of functions \( u_{p_i} : D_p \to D_p \) defined as follows:

\[
\begin{align*}
\text{put}_i ((d, \text{false})) & \triangleq (u_i(d), \text{false}) \\
\text{put}_i ((d, \text{true})) & \triangleq \\
& \begin{cases} 
(d, \text{true}) & \text{if } u_i(d) = d \\
(\top, \text{false}) & \text{otherwise}
\end{cases}
\end{align*}
\]
Because every set \( U \) of update operations implicitly contains the identity function, the same is true for the set \( U_p \) of state update operations. Furthermore, it is easy to show that state update operations commute, just as update operations do; that is, \( \forall d, i, j. ~ u_{p_i}(u_{p_j}(p)) = u_{p_j}(u_{p_i}(p)) \). LK: I’ve done the proof of this, but I’m not including it here because it seems pretty obvious from the definition...if anyone wants me to actually include it, then I will.

3.2.3. Stores. During the evaluation of \( \lambda_{LVish} \) 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 \( \text{Loc} \), to its state, which is some pair \((d, frz)\) from the set \( D_p \). The way that stores are handled in \( \lambda_{LVish} \) is very similar to how they are handled in \( \lambda_{LVar} \), 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 3.5 (store, \( \lambda_{LVish} \)). A store is either a finite partial mapping \( S : \text{Loc} \rightarrow (D_p - \{\top\}) \), or the distinguished element \( \top \). I use the notation \( S[l \mapsto (d, frz)] \) to denote extending \( S \) with a binding from \( l \) to \((d, frz)\). If \( l \in \text{dom}(S) \), then \( S[l \mapsto (d, frz)] \) 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 \([l_1 \mapsto (d_1, frz_1), l_2 \mapsto (d_2, frz_2), \ldots] \).

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

Definition 3.6 (store ordering, \( \lambda_{LVish} \)). A store \( S \) is less than or equal to a store \( S' \) (written \( S \sqsubseteq S' \)) iff:

- \( S' = \top_S \), or

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

Definition 3.7 (lub of stores, \( \lambda_{LVish} \)). The lub of two stores \( S_1 \) and \( S_2 \) (written \( S_1 \sqcup S_2 \)) is defined as follows:

- \( S_1 \sqcup S_2 = \top_S \) iff \( S_1 = \top_S \) or \( S_2 = \top_S \).
3. QUASI-DETERMINISTIC AND EVENT-DRIVEN PROGRAMMING WITH LVARS

- $S_1 \sqcup_S S_2 = \top_S$ iff there exists some $l \in \text{dom}(S_1) \cap \text{dom}(S_2)$ such that $S_1(l) \sqcup p S_2(l) = \top_p$.
- Otherwise, $S_1 \sqcup S_2$ is the store $S$ such that:
  - $\text{dom}(S) = \text{dom}(S_1) \cup \text{dom}(S_2)$, and
  - For all $l \in \text{dom}(S)$:
    $$S(l) = \begin{cases} 
      S_1(l) \sqcup_p S_2(l) & \text{if } l \in \text{dom}(S_1) \cap \text{dom}(S_2) \\
      S_1(l) & \text{if } l \notin \text{dom}(S_2) \\
      S_2(l) & \text{if } l \notin \text{dom}(S_1)
    \end{cases}$$

If, for example,
$$(d_1, \text{frz}_1) \sqcup_p (d_2, \text{frz}_2) = \top_p,$$
then
$$[l \mapsto (d_1, \text{frz}_1)] \sqcup_S [l \mapsto (d_2, \text{frz}_2)] = \top_S.$$ 

Just as a store containing a binding $l \mapsto \top$ can never arise during the execution of a $\lambda_{\text{LVar}}$ program, a store containing a binding $l \mapsto (\top, \text{frz})$ can never arise during the execution of a $\lambda_{\text{LVish}}$ 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 $D_p$.

3.2.4. $\lambda_{\text{LVish}}$: syntax and semantics. The syntax of $\lambda_{\text{LVish}}$ appears in Figure 3.1, and Figures 3.2 and 3.3 together give the operational semantics. As with $\lambda_{\text{LVar}}$ 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 2.3, and a context semantics, shown in Figure 2.4. The reduction semantics is also parameterized by the set $U$ of update operations.

The $\lambda_{\text{LVish}}$ grammar has most of the expression forms of $\lambda_{\text{LVar}}$: variables, values, application expressions, get expressions, and new. Instead of put expressions, it has puti expressions, which are the interface to the specified set of update operations. $\lambda_{\text{LVish}}$ also adds two new language forms, the freeze expression and the freeze — after — with expression, which I discuss in more detail below.
Given a lattice \((D, \sqsubseteq, \bot, \top)\) with elements \(d \in D\):

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

Figure 3.1. Syntax for \(\lambda_{\text{LVish}}\).

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

As with \(\lambda_{\text{LVar}}\), the \(\lambda_{\text{LVish}}\) context relation \(\mapsto\) 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 \(\lambda_{\text{LVar}}\), although the set of evaluation contexts that the metavariable \(E\) ranges over is different.
Given a lattice \((D, \sqsubseteq, \bot, \top)\) with elements \(d \in D\), and a set of \(U\) of update operations \(u_i : D \to D\):

\[
\text{incomp}(P) \triangleq \forall p_1, p_2 \in P. (p_1 \neq p_2 \implies p_1 \sqcup p_2 = \top_P)
\]

<table>
<thead>
<tr>
<th>Rule</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>E-Beta</td>
<td>(\langle S; (\lambda x. e) v \rangle \longrightarrow \langle S; e[x := v] \rangle)</td>
</tr>
<tr>
<td>E-New</td>
<td>(\langle S; \text{new} \rangle \longrightarrow \langle S[l \mapsto (\bot, \text{false})]; l \rangle)</td>
</tr>
</tbody>
</table>
| E-Put      | \(\begin{cases} 
\langle S; \text{put}_i l \rangle \longrightarrow \langle S[l \mapsto u_p(p_1)]; \bot \rangle & \text{if } S(l) = p_1 \text{ and } u_p(p_1) \neq \top_p \\
\langle S; \text{put}_i l \rangle \longrightarrow \text{error} & \text{if } S(l) = p_1 \text{ and } u_p(p_1) = \top_p 
\end{cases}\) |
| E-Get      | \(\langle S; \text{get} l P \rangle \longrightarrow \langle S; p_2 \rangle\) |
| E-Freeze-Init | \(\langle S; \text{freeze } l \text{ after } Q \text{ with } \lambda x. e \rangle \longrightarrow \langle S; \text{freeze } l \text{ after } Q \text{ with } \lambda x. e, \{\}, \{\} \rangle\) |
| E-Spawn-Handler | \(\begin{cases} 
\langle S; \text{spawn } l \text{ after } Q \text{ with } \lambda x. e, \{\}, \{\} \rangle \longrightarrow \langle S; \text{spawn } l \text{ after } Q \text{ with } \lambda x. e, \{d_1\}, \{d_2\} \rangle & \text{if } S(l) = (d_1, \text{frz}_1) \text{ and } d_2 \sqsubseteq d_1 \text{ and } d_2 \notin H \\
\langle S; \text{freeze } l \text{ after } Q \text{ with } \lambda x. e, \{d_1\}, \{d_2\} \cup H \rangle \longrightarrow \langle S; \text{freeze } l \text{ after } Q \text{ with } \lambda x. e_0, \{e_0[x := d_2], e, \ldots\}, \{d_2\} \cup H \rangle & \text{if } S(l) = (d_1, \text{frz}_1) \text{ and } d_2 \sqsubseteq d_1 \text{ and } d_2 \notin H \\
\langle S; \text{freeze } l \text{ after } Q \text{ with } \lambda x. e, \{v, \ldots\}, H \rangle \longrightarrow \langle S[l \mapsto (d_1, \text{true})]; d_1 \rangle & \text{if } S(l) = (d_1, \text{frz}_1) \\
\langle S; \text{freeze } l \rangle \longrightarrow \langle S[l \mapsto (d_1, \text{true})]; d_1 \rangle & \text{if } S(l) = (d_1, \text{frz}_1)
\end{cases}\) |

Figure 3.2. Reduction semantics for \(\lambda_{\text{LVish}}\).
3.2.5. 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 $\lambda_{\text{LVar}}$:

- **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 $\text{incomp}(P)$. 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 $p_2$.

$\lambda_{\text{LVish}}$ replaces the $\lambda_{\text{LVar}}$ put operation with the put_i operation, which is actually a set of operations that are the interface to the provided update operations $u_i$. For each update operation $u_i$, put_i (implemented by the E-Put rule) takes a pointer to an LVar and updates the LVar’s state to the 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.

3.2.6. Freezing and the $\text{freeze} \leftarrow \text{after} \rightarrow \text{with}$ primitive. The E-Freeze-Init, E-Spawn-Handler, E-Freeze-Final, and E-Freeze-Simple rules are all new additions to $\lambda_{\text{LVish}}$. 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` primitive, which models the “freeze-after” pattern I described in Section 3.1.3. The expression

\[
\text{freeze } l v \text{ after } Q \text{ with } f
\]

has the following semantics:

- It attaches the callback \( f \) to the LVar \( lv \). The callback will be executed, once, for each element of the event set \( Q \) that the LVar’s state reaches or surpasses. The callback is a function that takes a lattice element as its argument. Its return value is ignored, so it runs solely for effect. For instance, a callback might itself do a `put` to the LVar to which it is attached, triggering yet more callbacks.
- If execution reaches a point where there are no more elements of \( Q \) left to handle and no callbacks still running, then we have reached a quiescent state, the LVar \( lv \) 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, \( \lambda_{\text{lvish}} \) includes an auxiliary form,

\[
\text{freeze } l \text{ after } Q \text{ with } \lambda x. e_0, \{ e, \ldots \}, H
\]

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 \( \{ e, \ldots \} \) is the set of running callbacks; and
- The set \( H \) (a subset of the lattice \( D \)) represents those values in \( Q \) for which callbacks have already been launched; we call \( H \) the “handled” set.
Due to $\lambda_{LVish}$’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 $\text{put}_i$ 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 $\text{put}_i$ updates that would have enabled additional callbacks will instead fault, causing the program to step to $\text{error}$.

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 $\text{freeze} - \text{after} - \text{with}$ 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 $\text{freeze} - \text{after} - \text{with}$, 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 4.2.5, our LVish library provides a special runParThenFreeze function that does so, and thereby guarantees full determinism.
3. QUASI-DETERMINISTIC AND EVENT-DRIVEN PROGRAMMING WITH LVARS

3.3. Proof of quasi-determinism for $\lambda_{LVish}$

In this section, I give a proof of quasi-determinism for $\lambda_{LVish}$ 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 $\sigma$ terminate in configurations $\sigma'$ and $\sigma''$, then either $\sigma'$ and $\sigma''$ are the same configuration (up to a permutation on locations), or one of them is error. As with the determinism proof for $\lambda_{LVar}$ in Section 2.5, quasi-determinism follows from a series of supporting lemmas. The basic structure of the proof follows that of the $\lambda_{LVar}$ determinism proof closely. However, instead of the Independence property that I showed for $\lambda_{LVar}$ (Lemma 2.5), here I prove a more general property, Generalized Independence (Lemma 3.7), that accounts for the presence of both freezing and arbitrary update operations in $\lambda_{LVish}$. Also, in the setting of $\lambda_{LVish}$, the Strong Local Confluence property (Lemma 2.8) becomes Strong Local Quasi-Confluence (Lemma 3.10), which allows the possibility of an error result, and the quasi-confluence lemmas that follow—Strong One-sided Quasi-Confluence (Lemma 3.11), Strong Quasi-Confluence (Lemma 2.10), and Quasi-Confluence (Lemma 2.11)—all follow this pattern as well.

3.3.1. Permutations and permutability. As with $\lambda_{LVar}$, the $\lambda_{LVish}$ language 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 3.8 (permutation, $\lambda_{LVish}$). A permutation is a function $\pi : Loc \rightarrow Loc$ such that:

1. it is invertible, that is, there is an inverse function $\pi^{-1} : Loc \rightarrow Loc$ 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 lift $\pi$ to apply expressions, stores, and configurations. Because expressions and stores are defined slightly differently in $\lambda_{LVish}$ than they are in $\lambda_{LVar}$, we must update our definitions of permutation of a store and permutation of an expression:

**Definition 3.9 (permutation of an expression, $\lambda_{LVish}$).** A permutation of an expression $e$ is a function $\pi$ defined as follows:

\[
\begin{align*}
\pi(x) & \triangleq x \\
\pi(\lambda x. e) & \triangleq \lambda x. \pi(e) \\
\pi(e_1 e_2) & \triangleq \pi(e_1) \pi(e_2) \\
\pi(\text{get } e_1 e_2) & \triangleq \text{get } \pi(e_1) \pi(e_2) \\
\pi(p) & \triangleq p \\
\pi(l) & \triangleq \pi(l) \\
\pi(P) & \triangleq P \\
\pi(Q) & \triangleq Q \\
\pi(\text{put}_i e) & \triangleq \text{put}_i \pi(e) \\
\pi(\text{new}) & \triangleq \text{new} \\
\pi(\text{freeze } e) & \triangleq \text{freeze } \pi(e) \\
\pi(\text{freeze } e_1 \text{ after } e_2 \text{ with } e_3) & \triangleq \text{freeze } \pi(e_1) \text{ after } \pi(e_2) \text{ with } \pi(e_3) \\
\pi(\text{freeze } l \text{ after } Q \text{ with } \lambda x. e, \{e, \ldots\}, H) & \triangleq \text{freeze } \pi(l) \text{ after } Q \text{ with } \lambda x. \pi(e), \{\pi(e), \ldots\}, H
\end{align*}
\]

**Definition 3.10 (permutation of a store, $\lambda_{LVish}$).** A permutation of a store $S$ is a function $\pi$ defined as follows:

\[
\begin{align*}
\pi(\top_S) & \triangleq \top_S \\
\pi([l_1 \mapsto p_1, \ldots, l_n \mapsto p_n]) & \triangleq [\pi(l_1) \mapsto p_1, \ldots, \pi(l_n) \mapsto p_n]
\end{align*}
\]

And the definition of permutation of a configuration is as it was before:
Definition 3.11 (permutation of a configuration, \( \lambda_{LVish} \)). A \textit{permutation} of a configuration \( \langle S; e \rangle \) is a function \( \pi \) defined as follows: if \( \langle S; e \rangle = \text{name} \), then \( \pi(\langle S; e \rangle) \triangleq \text{name} \); otherwise, \( \pi(\langle S; e \rangle) \triangleq \langle \pi(S); \pi(e) \rangle \).

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

Lemma 3.3 (Permutability, \( \lambda_{LVish} \)). For any finite permutation \( \pi \),

\begin{enumerate}
  \item \( \sigma \leftrightarrow \sigma' \) if and only if \( \pi(\sigma) \leftrightarrow \pi(\sigma') \).
  \item \( \sigma \leftrightarrow \sigma' \) if and only if \( \pi(\sigma) \leftrightarrow \pi(\sigma') \).
\end{enumerate}

Proof. Similar to the proof of Lemma 2.1 (Permutability for \( \lambda_{LVar} \)); see Section A.11. 

3.3.2. Internal Determinism. As we did for \( \lambda_{LVar} \), we can prove that \( \lambda_{LVish} \)'s reduction semantics is internally deterministic: that is, if a configuration can step by the reduction semantics, there is only one rule by which it can step, and only one configuration to which it can step, modulo location names. This fact will be useful to us later on in the proof of Strong Local Quasi-Confluence (Lemma 3.10).

Lemma 3.4 (Internal Determinism, \( \lambda_{LVish} \)). If \( \sigma \leftrightarrow \sigma' \) and \( \sigma \leftrightarrow \sigma'' \), then there is a permutation \( \pi \) such that \( \sigma' = \pi(\sigma'') \).

Proof. Straightforward by cases on the rule of the reduction semantics by which \( \sigma \) steps to \( \sigma' \); the only interesting case is for the E-New rule. See Section A.12.

3.3.3. Locality. Just as with the determinism proof for \( \lambda_{LVar} \), proving quasi-determinism for \( \lambda_{LVish} \) 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: \( \langle S; E_1[e_1] \rangle \leftrightarrow \langle S_1; E_1[e'_1] \rangle \), and \( \langle S; E_2[e_2] \rangle \leftrightarrow \langle S_2; E_2[e'_2] \rangle \).
The Locality lemma says that \( \rightsquigarrow \) 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 \( \lambda_{\text{lvish}} \)), but the proof must account for the set of possible evaluation contexts in \( \lambda_{\text{lvish}} \) being different (and larger).

Lemma 3.5 (Locality, \( \lambda_{\text{lvish}} \)). If \( \langle S ; E_1[e_1] \rangle \rightsquigarrow \langle S_1; E_1[e'_1] \rangle \) and \( \langle S; E_2[e_2] \rangle \rightsquigarrow \langle S_2; E_2[e'_2] \rangle \) and \( E_1[e_1] = E_2[e_2] \), where \( E_1 \neq E_2 \), 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.13.

3.3.4. Monotonicity. The Monotonicity lemma says that, as evaluation proceeds according to the \( \rightsquigarrow \) relation, the store can only grow with respect to the \( \sqsubseteq_S \) ordering.

Lemma 3.6 (Monotonicity, \( \lambda_{\text{lvish}} \)). If \( \langle S; e \rangle \rightsquigarrow \langle S'; e' \rangle \), 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.14.

3.3.5. Generalized Independence. Recall from Chapter 2 that in order to prove determinism for \( \lambda_{\text{lvish}} \), we needed to establish a “frame property” that captures the idea that independent effects commute with each other. For \( \lambda_{\text{lvish}} \), the Independence lemma (Lemma 2.5) established that property. It shows that, if a configuration \( \langle S; e \rangle \) can step to \( \langle S'; e' \rangle \), then it is possible to “frame on” an additional store \( S'' \) without interfering with the ability to take a step—that is, \( \langle S \sqcup_S S''; e \rangle \rightsquigarrow \langle S' \sqcup_S S''; e' \rangle \), subject to certain restrictions on \( S'' \).
3. QUASI-DETERMINISTIC AND EVENT-DRIVEN PROGRAMMING WITH LVARS

For \( \lambda_{LVish} \), we need to establish a similar frame property. However, since we have generalized from `put` to `put\_i`, we also need to generalize our frame property. In fact, the original Lemma 2.5 does not hold for \( \lambda_{LVish} \). As an example, consider an LVar whose states form a lattice \( \bot < 0 < 1 < \top \). Consider the transition

\[
\langle [l \mapsto (0, \text{false})] ; \text{put\_i}\rangle \longleftarrow \langle [l \mapsto (1, \text{false})] ; () \rangle,
\]

where the update operation \( u_i \) happens to increment its argument by one. Now suppose that we wish to “frame” the store \( [l \mapsto (1, \text{false})] \) onto this transition using Lemma 2.5; that is, we wish to show that

\[
\langle [l \mapsto (0, \text{false})] \cup_S [l \mapsto (1, \text{false})] ; \text{put\_i}\rangle \longleftarrow \langle [l \mapsto (1, \text{false})] \cup_S [l \mapsto (1, \text{false})] ; () \rangle.
\]

We know that \( [l \mapsto (1, \text{false})] \cup_S [l \mapsto (1, \text{false})] \neq \top_S \), which is required to be able to apply Lemma 2.5. Furthermore, \( [l \mapsto (1, \text{false})] \) is non-conflicting with the original transition, since no new locations are allocated between \( [l \mapsto (0, \text{false})] \) and \( [l \mapsto (1, \text{false})] \). However, it is not the case that \( \langle [l \mapsto (0, \text{false})] \cup_S [l \mapsto (1, \text{false})] ; \text{put\_i}\rangle \) steps to \( \langle [l \mapsto (1, \text{false})] \cup_S [l \mapsto (1, \text{false})] ; () \rangle \), because \( u_{p_i}((S \cup_S S'')(l)) = \top_p \). (As before, \( u_{p_i} \) is the update operation \( u_i \), lifted from lattice elements \( d \) to states \( (d, frz) \).)

What went wrong here? The problem is that, as previously discussed in Section 2.6.1, lub operations do not necessarily commute with arbitrary update operations. In \( \lambda_{LVar} \), where the only “update operation” is a lub write performed via `put`, it is fine that the Independence lemma uses a lub operation to frame \( S'' \) onto the transition. For \( \lambda_{LVish} \), though, we need to state our frame property in a way that will allow it to accommodate any update operation from the given set \( U \).

Therefore, I define a store update operation \( U_S \) to be a function from stores to stores that can add new bindings, update the contents of existing locations using operations \( u_i \) from the given set \( U \) of update operations (or, more specifically, their lifted versions \( u_{p_i} \)), or freeze the contents of existing locations.

Definition 3.12 (store update operation). Given a lattice \( (D, \sqsubseteq, \top, \bot) \) and a set of state update operations \( U_p \), a store update operation is a function \( U_S \) from stores to stores such that:
\[ \text{dom}(U_S(S)) \supseteq \text{dom}(S); \]

\[ \text{for each } l \in \text{dom}(S), \text{ either:} \]

\[ \begin{align*}
&\text{– } (U_S(S))(l) = u_{p_i}(S(l)), \text{ where } u_{p_i} \in U_p, \text{ or} \\
&\text{– } (U_S(S))(l) = (d, \text{true}), \text{ where } S(l) = (d, \text{frz}); \text{ and}
\end{align*} \]

\[ \text{for each } l \in \text{dom}(U_S(S)) \text{ that is not a member of } \text{dom}(S), (U_S(S))(l) = (d, \text{frz}) \text{ for some } d \in D. \]

Definition 3.12 says that applying \( U_S \) to \( S \) either updates (using some \( u_{p_i} \in U_p \)) or freezes the contents of each \( l \in \text{dom}(S) \). Since the identity function is always implicitly a member of \( U_p \), \( U_S \) can act as the identity on the contents of locations. \( U_S \) can also add new bindings to the store it operates on; however, it cannot change existing location names.

With Definition 3.12 in hand, we can state a more general version of the Independence lemma:

**Lemma 3.7 (Generalized Independence).** If \( \langle S; e \rangle \xrightarrow{\text{error}} \langle S'; e' \rangle \) \((\text{where } \langle S'; e' \rangle \neq \text{error})\), then we have that:

\[ \langle U_S(S); e \rangle \xrightarrow{\text{error}} \langle U_S(S'); e' \rangle, \]

where \( U_S \) is a store update operation meeting the following conditions:

\[ \begin{align*}
&\text{– } U_S \text{ is non-conflicting with } \langle S; e \rangle \xrightarrow{\text{error}} \langle S'; e' \rangle, \\
&\text{– } U_S(S') \neq \top_S, \text{ and} \\
&\text{– } U_S \text{ is freeze-safe with } \langle S; e \rangle \xrightarrow{\text{error}} \langle S'; e' \rangle. 
\end{align*} \]

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, E-Put, E-Freeze-Final, and E-Freeze-Simple rules. See Section A.15. \( \Box \)

Lemma 3.7 has three preconditions on the store update operation \( U_S \), two of which mirror the two preconditions on \( S'' \) from the original Independence lemma: the requirement that \( U_S(S') \neq \top_S \), and the requirement that \( U_S \) is non-conflicting with the transition from \( \langle S; e \rangle \) to \( \langle S'; e' \rangle \). Definition 3.13 revises our previous definition of “non-conflicting” to apply to store update operations. It says that
Definition 3.13 (non-conflicting store update operation). A store update operation $U_S$ is non-conflicting with the transition $\langle S; e \rangle \rightarrow \langle S'; e' \rangle$ if, for all locations allocated during the transition, $U_S$ does not interfere with those locations. For instance, if $l$ is allocated in the transition from $\langle S; e \rangle$ to $\langle S'; e' \rangle$, then $l \notin \text{dom}(U_S(S))$ (that is, $U_S$ cannot add a binding at $l$ to $S$), and $(U_S(S'))(l) = S'(l)$ (that is, $U_S$ cannot update the contents of $l$ in $S'$).

The third precondition on $U_S$ has to do with freezing: $U_S$ must be freeze-safe with the transition $\langle S; e \rangle \rightarrow \langle S'; e' \rangle$, which means that, for any locations that change in status (that is, become frozen) during the transition, $U_S$ cannot update the contents of those locations. This precondition is only needed in the E-Freeze-Final and E-Freeze-Simple cases, and it has the effect of ruling out interference from freezing. (Note that $U_S$ need not avoid updating the contents of locations that are already frozen before the transition takes place. This corresponds to the fact that, if an LVar is already frozen, arbitrary updates to it do, in fact, commute with freeze operations on it—those later freeze operations will have no effect, and updates will either have no effect or raise an error.)

Definition 3.14 (freeze-safe store update operation). A store update operation $U_S$ is freeze-safe with the transition $\langle S; e \rangle \rightarrow \langle S'; e' \rangle$ iff, for all $l \in (\text{dom}(S') - \text{dom}(S))$, $U_S$ neither creates new bindings at $l$ nor updates existing bindings at $l$. The two changes we have made to the Independence lemma—the use of $U_S$, and the requirement that $U_S$ be freeze-safe with the transition in question—are orthogonal to each other, in accordance with the fact that arbitrary update operations are an orthogonal language feature to freezing. A version of $\lambda_{LVish}$ that had freezing, but retained the lub semantics of put in $\lambda_{LVar}$, could use the old formulation of the Independence lemma, taking the lub of the original stores and a frame store $S''$, but it would
still need to have a requirement on \( S'' \) to rule out interference from freezing. On the other hand, a version of the language \( \text{without} \) freezing, but \( \text{with} \) arbitrary updates, would still use \( U_S \) but could leave out the requirement that it be freeze-safe (since the requirement would be vacuously true anyway). I make particular note of the orthogonality of freezing and arbitrary updates because freezing introduces quasi-determinism, while arbitrary updates do not.\(^4\)

Finally, although it no longer uses an explicit “frame” store, we can still think of Lemma 3.7 as a frame property; in fact, it is reminiscent of the generalized frame rule of the “Views” framework [18], which I discuss in more detail in Section 6.6.

3.3.6. Generalized Clash. The Generalized Clash lemma, Lemma 3.8, is similar to the Generalized Independence lemma, but handles the case where \( U_S(S') = \top_S \). It establishes that, in that case, \( \langle U_S(S); e \rangle \) steps to error in at most one step.

Lemma 3.8 (Generalized Clash). If \( \langle S; e \rangle \leadsto \langle S'; e' \rangle \) (where \( \langle S'; e' \rangle \neq \text{error} \)), then we have that:

\[
\langle U_S(S); e \rangle \leadsto^i \text{error}, \quad \text{where } i \leq 1,
\]

and where \( U_S \) is a store update operation meeting the following conditions:

- \( U_S \) is non-conflicting with \( \langle S; e \rangle \leadsto \langle S'; e' \rangle \),
- \( U_S(S') = \top_S \), and
- \( U_S \) is freeze-safe with \( \langle S; e \rangle \leadsto \langle S'; e' \rangle \).

Proof. By cases on the rule of the reduction semantics by which \( \langle S; e \rangle \) steps to \( \langle S'; e' \rangle \). Since \( \langle S'; e' \rangle \neq \text{error} \), we do not need to consider the E-Put-Err rule. See Section A.16.\(\square\)

\(^4\)To rigorously show that arbitrary updates retain full determinism and not merely quasi-determinism, I would need to define yet another language, one that generalizes \text{put} to \text{put}_i but does not introduce freezing, and then prove determinism for \textbf{that} language. Instead, I hope to informally convince you that the quasi-determinism in \text{LVish} comes from freezing, rather than from arbitrary updates.
3. QUASI-DETERMINISTIC AND EVENT-DRIVEN PROGRAMMING WITH LVARS

3.3.7. Error Preservation. Lemma 3.9, Error Preservation, is the $\lambda_{LVish}$ counterpart of Lemma 2.7 from Chapter 2. It 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 3.9 (Error Preservation, $\lambda_{LVish}$). If $\langle S; e \rangle \rightarrow \text{error}$ and $S \sqsubseteq S'$, then $\langle S'; e \rangle \rightarrow \text{error}$.

Proof. Suppose $\langle S; e \rangle \rightarrow \text{error}$ and $S \sqsubseteq S'$. We are required to show that $\langle S'; e \rangle \rightarrow \text{error}$.

By inspection of the operational semantics, the only rule by which $\langle S; e \rangle$ can step to error is E-Put-Err. Hence $e = \text{put}_i l$. From the premises of E-Put-Err, we have that $S(l) = p_1$. Since $S \sqsubseteq S'$, it must be the case that $S'(l) = p'_1$, where $p_1 \sqsubseteq p'_1$. Since $u_{p_1}(p_1) = \top_p$, we have that $u_{p'_1}(p'_1) = \top_p$. Hence, by E-Put-Err, $\langle S'; \text{put}_i l \rangle \rightarrow \text{error}$, as we were required to show. □

3.3.8. Quasi-Confluence. Lemma 3.10 says that if a configuration $\sigma$ can step to configurations $\sigma_a$ and $\sigma_b$, then one of two possibilities is true: either there exists a configuration $\sigma_c$ that $\sigma_a$ and $\sigma_b$ can each reach in at most one step, modulo a permutation on locations, or at least one of $\sigma_a$ or $\sigma_b$ steps to error. Lemmas 3.11 and 3.12 then generalize that result to arbitrary numbers of steps.

Lemma 3.10 (Strong Local Quasi-Confluence). If $\sigma \rightarrow \sigma_a$ and $\sigma \rightarrow \sigma_b$, then either:

1. there exist $\sigma_c, i, j, \pi$ such that $\sigma_a \rightarrow^i \sigma_c$ and $\pi(\sigma_b) \rightarrow^j \sigma_c$ and $i \leq 1$ and $j \leq 1$, or
2. $\sigma_a \rightarrow \text{error}$ or $\sigma_b \rightarrow \text{error}$.

Proof. As in the proof of Strong Local Confluence for $\lambda_{LVar}$ (Lemma 2.8), since the original configuration $\sigma$ 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. In the special case where $E_a = E_b$, the result follows by Internal Determinism (Lemma 3.4).

If $E_a \neq E_b$, we can apply the Locality lemma (Lemma 3.5); at a high level, it shows that $e_{a_1}$ and $e_{b_1}$ can be evaluated independently within their contexts. The proof is then by a double case analysis on
the rules of the reduction semantics by which \( \langle S; e_a \rangle \) steps and by which \( \langle S; e_b \rangle \) steps. In order to combine the results of the two independent steps, the proof makes use of the Generalized Independence lemma 3.7. In almost every case, there does exist a \( \sigma_c \) to which \( \sigma_a \) and \( \sigma_b \) both step; the only cases in which we need to resort to the \textbf{error} possibility are those in which one step is by E-Put and the other is by E-Freeze-Final or E-Freeze-Simple—that is, the situations in which a write-after-freeze error is possible. See Section A.17.

Lemma 3.11 (Strong One-Sided Quasi-Confluence). If \( \sigma \rightarrow \sigma' \) and \( \sigma \rightarrow^m \sigma'' \), where \( 1 \leq m \), then either:

1. there exist \( \sigma_c, i, j, \pi \) such that \( \sigma' \rightarrow^i \sigma_c \) and \( \pi(\sigma'') \rightarrow^j \sigma_c \) and \( i \leq m \) and \( j \leq 1 \), or
2. there exists \( k \leq m \) such that \( \sigma' \rightarrow^k \text{error} \), or there exists \( k \leq 1 \) such that \( \sigma'' \rightarrow^k \text{error} \).

Proof. By induction on \( m \); see Section A.18.

Lemma 3.12 (Strong Quasi-Confluence). If \( \sigma \rightarrow \gamma^m \sigma' \) and \( \sigma \rightarrow \gamma^m \sigma'' \), where \( 1 \leq n \) and \( 1 \leq m \), then either:

1. there exist \( \sigma_c, i, j, \pi \) such that \( \sigma' \rightarrow^i \sigma_c \) and \( \pi(\sigma'') \rightarrow^j \sigma_c \) and \( i \leq m \) and \( j \leq n \), or
2. there exists \( k \leq m \) such that \( \sigma' \rightarrow^k \text{error} \), or there exists \( k \leq n \) such that \( \sigma'' \rightarrow^k \text{error} \).

Proof. By induction on \( n \); see Section A.19.

Lemma 3.13 (Quasi-Confluence). If \( \sigma \rightarrow \gamma^* \sigma' \) and \( \sigma \rightarrow \gamma^* \sigma'' \), then either:

1. there exist \( \sigma_c, \pi \) such that \( \sigma' \rightarrow^* \sigma_c \) and \( \pi(\sigma'') \rightarrow^* \sigma_c \), or
2. \( \sigma' = \text{error} \) or \( \sigma'' = \text{error} \).

3.3.9. **Quasi-Determinism.** The Quasi-Determinism theorem, Theorem 3.1, is a straightforward result of Lemma 3.13. It says that if two executions starting from a configuration $\sigma$ terminate in configurations $\sigma'$ and $\sigma''$, then $\sigma'$ and $\sigma''$ are the same configuration, or one of them is error.

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

1. there exists $\pi$ such that $\sigma' = \pi(\sigma'')$, or
2. $\sigma' = \text{error}$ or $\sigma'' = \text{error}$.

Proof. By Lemma 3.13, one of the following two cases applies:

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

   By Lemma 3.3 (Permutability), $\sigma''$ can step iff $\pi(\sigma'')$ can step, so since $\sigma''$ 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. $\sigma' = \text{error}$ or $\sigma'' = \text{error}$, and so the result is immediate.

3.3.10. **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 $\lambda_{LVish}$ 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 $\rightarrow$ after $\rightarrow$ primitive, but even those that freeze in arbitrary places using the simpler freeze $\rightarrow$ 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 possibility 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 race.

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.
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 [39], 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 of Chapter 2, those effects are put and get operations on LVars; Chapter 3 adds the freeze operation and arbitrary update operations. Implementing these operations as monadic effects in Haskell makes it possible to provide compile-time guarantees about determinism and quasi-determinism, because as long as the only monad that programs use is the one in which LVars operations are allowed to run, we know that they can only perform the side effects that we have chosen to allow.

Another reason why the existing Par monad is an appealing conceptual starting point for a practical implementation of LVars is that it already allows inter-task communication through IVars, which, as we have seen, are a special case of LVars. 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.

In this chapter, I describe the LVish library, a Haskell library for practical deterministic and quasi-deterministic parallel programming with LVars. We have already seen 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 more thorough tour of LVish.

4.1. The big picture

Our library adopts and builds on the basic approach of the Par monad and the monad-par library [39], 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 update operations, get, freeze, and event handlers.

Yet our library also includes 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. Control operators like fork are the only non-data-structure-specific operations exposed to application writers.

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 [29]. Data structures that fit into the L Vish 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 can be simpler to represent and implement than general ones.

4.2. The L Vish library interface for application writers

In this section I illustrate the use of the L Vish library from the point of view of the application writer, through a series of short example programs.  

4.2.1. A simple example: IVars in L Vish. As mentioned in the previous section, the L Vish 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. Listing 4.1 shows a program written using monad-par that will deterministically raise an error, because it tries to write to the IVar num twice. 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 twice, via two calls to put, each of which runs in a separately forked task. The runPar function is an implicit global barrier: all forks have to complete before runPar can return.

---

2The examples in this section, among others, are available at https://github.com/lkuper/lvar-examples/.
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)

Listing 4.1. A basic IVar example using monad-par.

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)

Listing 4.2. Repeated writes of the same value to an IVar.

Listing 4.1 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. (Since we are using monad-par and not LVish, 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, when using monad-par, even multiple writes of the same value to an IVar will raise a “multiple put” error, as in Listing 4.2. This program differs from the previous one only in that the two puts 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 nevertheless raises an error because of monad-par’s single-write restriction on IVars.

Now let us consider a version of Listing 4.2 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
### 4. THE LVISH LIBRARY: INTERFACE, IMPLEMENTATION, AND EVALUATION

```haskell
{-# 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)
```

Listing 4.3. Repeated writes of the same value to an LVar.

on to consider some more sophisticated LVars later in this section.) Listing 4.3 shows an LVish program that will write 4 to an IVar twice and then deterministically print 4 instead of raising an error.

In Listing 4.3, we need to 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 computes a lub (with respect to a lattice like that of Figure 2.1(b), except including all the Ints), and the `get` operation performs a threshold read, where the threshold set is implicitly the set of all Ints. 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 lub 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 a LANGUAGE pragma,
instructing the compiler that we are now using the TypeFamilies language extension. 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 deterministic and quasi-deterministic programming in LVish, we need a way to specify which 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. LVish follows the precedent of Kiselyov et al. on extensible effects in Haskell [33]: 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 approach 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 TypeFamilies language extension in our LVish programs. For brevity, I will elide the LANGUAGE pragmas in the rest of the example LVish programs in this section.
The LVish Par type constructor also has a second type parameter, s, making \( \text{Par} \in s \) 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. An observably deterministic shopping cart

For our next few examples, let us consider concurrently adding items to a shopping cart. Suppose we have an \( \text{Item} \) 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:

```haskell
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. The name IMap is by analogy with IVar, but here, it is individual entries in the map that are immutable, not the map itself. If a key is inserted multiple times, the values must be equal (according to \(==\)), or a “multiple put” error will be raised.

\(^3\)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” [37].

\(^4\)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, and either implementation of IMap would have worked for this example, 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.
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)

Listing 4.4. A deterministic shopping cart program.

Listing 4.4 shows an LVish program that inserts items into our cart. 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 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.

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 `IMap` 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 with LVars, because it would go against the principle of monotonic growth.\footnote{On the other hand, one way to implement 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. \textit{Conflict-free replicated data types} (CRDTs) \cite{[54]} use variations on this approach to implement various data structures that support seemingly non-monotonic operations. I discuss the relationship of LVars to CRDTs in more detail in Chapter 5.}
import Control.LVish
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)

Listing 4.5. A quasi-deterministic shopping cart program.

4.2.4. A quasi-deterministic shopping cart. The LVish examples we have seen so far 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 adding items to it.

In Listing 4.5, we are inserting items into our cart, as in Listing 4.4. But, instead of returning the result of a 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 Items to Ints. 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, instead of HasGet, we have the 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 of runPar, called runParQuasiDet, whose type signature allows Par computations that allow freezing to be passed to it.

The quasi-determinism in Listing 4.5 arises from the fact that the call to freezeMap may run before both forked computations have completed. In this example, one or both calls to insert may run after...
import Control.LVish
import Control.LVish.DeepFrz -- provides runParThenFreeze
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)

Listing 4.6. A deterministic shopping cart program that uses runParThenFreeze.

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.

4.2.5. 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 the 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 manually calling freeze (and perhaps accidentally freezing an LVar too early), we can tell LVish to handle the freezing for us while “on the way out” of a Par computation. The mechanism that allows this is another variant of runPar, which we call runParThenFreeze.

Listing 4.6 shows a version of Listing 4.5 written using runParThenFreeze. An interesting thing to note about the Par computation in Listing 4.6 is that it only performs writes (as we can see from its
effect level, which is only constrained by \texttt{HasPut}). Also, unlike in Listing 4.5, where the \texttt{freeze} took place inside the \texttt{Par} computation, in Listing 4.6 the \texttt{Par} computation returns an \texttt{IMap} rather than a \texttt{Map}. Since \texttt{IMap} is an \texttt{LVar}, it has an \texttt{s} parameter, which we can see in the type of \texttt{p}.

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

4.2.6. Event-driven programming with \texttt{LVars}: a deterministic parallel graph traversal. Finally, let us look at an example that uses event handlers as well as freezing. In Listing 4.7, the function \texttt{traverse} takes a graph \texttt{g} and a vertex \texttt{startNode} and finds the set of all vertices reachable from \texttt{startNode}, in parallel. The \texttt{traverse} function first creates a new \texttt{LVar}, called \texttt{seen}, to represent a monotonically growing set of \texttt{Int}s that will identify nodes in the graph. For this purpose, we use the \texttt{ISet} type, provided by the \texttt{Data.LVar.PureSet} module. (As with \texttt{IMap}, the individual elements of the \texttt{ISet} are immutable, but the set itself can grow.)

Next, \texttt{traverse} attaches an event handler to \texttt{seen}. It does so by calling the \texttt{newHandler} function, which takes two arguments: an \texttt{LVar} and the callback that is to be to run every time an event occurs on that \texttt{LVar} (in this case, every time a new node is added to the set).\footnote{\texttt{newHandler} is not provided by LVish, but we can easily implement it using LVish’s built-in \texttt{newPool} and \texttt{addHandler} operations. LK: Actually, is there any good reason why LVish doesn’t provide something like \texttt{newHandler}?} The callback responds to events by looking up the neighbors of the newly arrived node (assuming a \texttt{neighbors} operation, which takes a graph and a vertex and returns a list of the vertex’s neighbor vertices), then mapping the \texttt{insert} function over that list of neighbors.

Finally, \texttt{traverse} adds the starting node to the \texttt{seen} set by calling \texttt{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 \texttt{quiesce h} returns; it will block until all events have been handled. Finally, we freeze and return the
import Control.LVish
import Control.LVish.DeepFrz -- provides Frzn
import Data.LVar.Generic (addHandler, freeze)
import Data.LVar.PureSet
import qualified Data.Graph as G

traverse :: (HasPut e, HasFreeze e) =>
        G.Graph -> Int -> Par e s (ISet Frzn Int)
traverse g startNode = do
  seen <- newEmptySet
  h <- newHandler seen
  (\node -> do
    mapM (\v -> 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)

Listing 4.7. A deterministic parallel graph traversal with an explicit call to freeze.

ISet of all reachable nodes. Since ISet is an LVar, it has an s parameter, and in the return type of
traverse, the s parameter of ISet has been replaced by the Frzn type, indicating that the LVar has
been frozen.

The good news is that this particular graph traversal program 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.

However, just as with our final shopping cart example in Listing 4.6, we can use runParThenFreeze to
ensure that freezing happens last. Listing 4.8 gives a version of traverse that uses runParThenFreeze,
and eliminates the possibility of forgetting to call quiesce and thereby introducing quasi-determinism:
4. THE LVISH LIBRARY: INTERFACE, IMPLEMENTATION, AND EVALUATION

import Control.LVish
import Control.LVish.DeepFrz -- provides runParThenFreeze
import Data.LVar.Generic (addHandler, freeze)
import Data.LVar.PureSet
import qualified Data.Graph as G

traverse :: HasPut e ⇒ G.Graph → Int → Par e s (ISet s Int)
traverse g startNode = do
  seen ← newEmptySet
  h ← newHandler seen
  \node → 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)))

Listing 4.8. A deterministic parallel graph traversal that uses runParThenFreeze.

In Listing 4.8, since freezing is performed by runParThenFreeze rather than an explicit call to freeze, it is no longer necessary to constrain e with HasFreeze in the type of traverse. Furthermore, the s parameter in the ISet that traverse returns can remain s instead of being instantiated with Frzn. Most importantly, since freezing is performed by runParThenFreeze rather than an explicit call to freeze, it is no longer necessary for traverse 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 effect level \( e \) 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, \( T \ (\text{Par} \ e \ s) a \), is a computation in that monad. Indeed, Par-monad transformers are valid monad transformers (in the sense of providing a standard MonadTrans instance).
4. THE LVISH LIBRARY: INTERFACE, IMPLEMENTATION, AND EVALUATION

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 :: a 
  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 [40].

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 must be

7Although 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.
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:

```haskell
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 (DPI)[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.

```plaintext
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, 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

---

8That 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

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 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 ⊥, but for which the only smaller element is ⊥. Lattices for which every element is the lub of some set of atoms are called atomistic, and in practice most lattices used by LVish programs have this property—especially those that represent collections. For example, the lattice of finite maps is atomistic, with the atoms consisting of all singleton maps, i.e., all key/value pairs.

For data structures whose states form an atomistic lattice, we usually want to expose operations that work at the atom level, semantically limiting writes to atoms, gets to threshold sets of atoms, and event sets to sets of atoms. For example, the interface to finite maps allows addition of a new key/value pair (an atom), querying of a single key, or traversals (which we model as handlers) that walk over one key/value pair at a time.

The LVish implementation is designed to facilitate good performance for data structures whose states form an atomistic lattice. We accomplish this by associating an LVar type with a set of deltas, or changes,
as well as a lattice. For atomistic lattices, the deltas are essentially just the atoms. "essentially just" seems weaselly — what does it mean? Any chance we can get rid of those two words?—for a set data structure, for instance, a delta is an element of the set; for a map, a delta is 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 writes and gets/handlers.

Of course, not every lattice that can be associated with an LVar type is atomistic. The lattice of Figure 2.1(a), for instance, is not atomistic and corresponds to an increment-only integer counter LVar. Therefore our implementation supports arbitrary lattices, not just atomistic ones.

4.4.2. Representation choices. LK: I think everything in this subsection is now up-to-date and accurate wrt the current LVish implementation, but it would be great if someone else would check it over.

LVish uses the following generic internal representation for LVars:

```haskell
data LVar a d = LVar { 
  state :: a, -- current, "global" state of LVar
  status :: {-# UNPACK #-} !(IORef (Status d)), -- is the LVar active or frozen?
  name :: {-# UNPACK #-} !LVarID -- a unique identifier for this LVar
}
```

where the type parameter a is the (mutable) data structure representing the lattice, and d is the type of deltas for the lattice. For a set data structure, for instance, a is the set type, but d is the type of an element of the set. For non-atomistic lattices, we take a and d to be the same type. For instance, for an increment-only counter, both a and d are Int. The status field is a mutable reference that represents the status bit, which says whether or not an LVar is frozen:

```haskell
data Status d 
  = Freezing          -- ^ further changes to the state are forbidden
  | Frozen            -- ^ further changes to the state are forbidden
  | Active (B.Bag (Listener d)) -- ^ bag of blocked threshold reads and handlers
```
The status bit of an LVar is tied together with a bag of waiting listeners, which include blocked gets and handlers; once the LVar is frozen, there can be no further events to listen for. The bag module (imported as B) supports atomic insertion and removal, and concurrent traversal:

```haskell
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:

```haskell
data Listener d = Listener {
  onUpdate :: d → B.Token (Listener d) → SchedState → IO (),
  onFreeze :: B.Token (Listener d) → SchedState → 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, SchedState, 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:

```haskell
type ClosedPar = SchedState → IO ()
type ParCont a = a → ClosedPar
mkPar :: (ParCont a → ClosedPar) → Par 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

9In particular, with one atomic update of the flag we both mark the LVar as frozen and allow the bag to be garbage-collected.
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. Listing 4.9 gives the implementation for the lattice-generic getLV function, which assists data structure authors in writing operations with get semantics. In addition to an LVar, it takes two threshold functions as arguments, one for global state and one for deltas.\footnote{In this sense, the way that threshold reads are implemented in LVish is in fact quite close to the semantic notion of \textit{threshold functions} that I described in Section 2.6.3.} 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 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:

```hs
-- 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 write 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
getLV :: (LVar a d) -> (a -> Bool -> IO (Maybe b))
        -> (d -> IO (Maybe b)) -> Par e s b
getLV (LVar {state, status}) gThresh dThresh =
    mkPar $ \k q -> do
    curStatus <- readIORef status
    case curStatus of
        Active listeners -> do
            tripped <- gThresh state False
            case tripped of
                Just b -> exec (k b) q -- fast path
                Nothing -> do
                    ifWinRace <- newRedupCheck (Sched.idemp q)
                    let onUpdate d = unblockWhen (dThresh d)
                        onFreeze = unblockWhen (gThresh state True)
                        unblockWhen thresh tok q = do
                            tripped <- thresh
                            whenJust tripped $ \b -> do
                                B.remove tok
                                ifWinRace (Sched.pushWork q (k b)) (return ()
                        tok <- B.put listeners (Listener onUpdate onFreeze)
                        frozen <- isFrozen status
                        tripped' <- gThresh state frozen
                        case tripped' of
                            Just b -> do
                                B.remove tok -- remove the listener we just added, and
                                ifWinRace (exec (k b) q) (sched q)
                                -- execute our continuation, or go back to the scheduler.
                                Nothing -> sched q
                            Nothing -> do
                                tripped <- gThresh state True
                                case tripped of
                                    Just b -> exec (k b) q -- already past the threshold; invoke the
                                    -- continuation immediately
                                    Nothing -> sched q -- We'll NEVER be above the threshold.
                        Frozen -> do
                            tripped <- gThresh state True
                            case tripped of
                                Just b -> exec (k b) q -- already past the threshold; invoke the
                                -- continuation immediately
                                Nothing -> sched q -- We'll NEVER be above the threshold.

Listing 4.9. Implementation of the getLV function.

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. But it may 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
putLV :: LVar a d → (a → IO (Maybe d)) → Par e s ()
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

Listing 4.10. Idempotence-assuming implementation of the putLV function. The version supporting non-idempotent (Sched.idemp==False), must also use handlerStatus as a reader-writer lock, except with put being the “reader” and addHandler as the writer.

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

Listing 4.11. Implementation of the freezeLV function.

delta). Whether this is a problem depends on whether the effect combination used in the Par session is idempotent. If Sched. idemp q is false—i.e. if non-idempotent update operations are used—then only one invocation of ifWinRace a b, can run a, all others must run b. This is a performance tradeoff: if all effects are idempotent, we avoid imposing extra synchronization on all uses of getLV at the cost of some duplicated work in a rare case.

4.4.4. Putting and freezing. TODO: I think this section needs to be updated to accommodate non-idempotent writes, but I need some help in figuring out what to say.
Listings 4.10 and 4.11 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.\footnote{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 [24].}

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
is extremely cheap. The only time that the busy flags can create cross-core communication is during
freezeLV, which should only occur once in each LVar’s lifetime.

One final point: unlike getLV and putLV, which can be used in deterministic contexts, a computation
that performs freezeLV is necessarily quasi-deterministic or non-deterministic.

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

```haskell
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.\footnote{One can use a high-performance scalable non-zero indicator\cite{20} to implement Counter. RN: Actually, it’s implemented
and passing tests, so let’s turn SNZI on.} 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 e s HandlerPool), and implement quiescence testing as
follows:

```haskell
quiesce :: HandlerPool -> Par 1lvl ()
quiesce hp@(HandlerPool cnt bag) = mkPar $ \k 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:

\[
\text{addHandler} :: \text{Maybe}\;\text{HandlerPool} \to \text{Pool} \to \text{LVar}\;a\;d \to (a \to \text{IO}\;\text{Maybe}\;\text{Par}\;\text{lvl}\;()) \to (d \to \text{IO}\;\text{Maybe}\;\text{Par}\;\text{lvl}\;()) \to \text{Global}\;\text{callback} \to \text{Delta}\;\text{callback} \to \text{Par}\;\text{lvl}\;()
\]

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: tracking and leveraging idempotency. While I have emphasized the commutativity of the lub operation, it also has another important property: idempotence, meaning that \(d \sqcup d = d\) for any element \(d\). In LVar terms, repeated puts or freezes have no effect, and so, in an LVish program that restricts itself to 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 [42]: 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:

```haskell
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:

```
LK: This is from the PLDI paper, but shouldn’t it also have `NoPut` as a constraint? RN: Why NoPut? Bumps disallow any optimizations that may duplicate work, but puts still work fine, right?
```

```
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 programming models, 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 [43]. The algorithm processes expressions written in a continuation-passing-style $\lambda$-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. Furthermore, 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 needing to actually run 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 one: breadth-first exploration. The `explore` function from the original, sequential version of the analysis, shown in Listing 4.12, expresses the heart of the search process. `explore` 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

---

13Haskell port by Max Bolingbroke: [https://github.com/batterseapower/haskell-kata/blob/master/0CFA.hs](https://github.com/batterseapower/haskell-kata/blob/master/0CFA.hs)
Listing 4.12. The `explore` function from a purely functional $\kappa$-CFA implementation.

```haskell
explore :: S.Set State \rightarrow [State] \rightarrow S.Set State
explore seen [] = seen
explore seen (todo:todos)
  | todo `S.member` seen = explore seen todos
  | otherwise = explore (S.insert todo seen) (S.toList (next todo) ++ todos)
```

with futures—for instance, using the Haskell Strategies library [38]. 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 `S.member` 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:

```haskell
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. $\kappa$-CFA phase two: 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 `Data.Set` or `Data.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 one and phase two 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 Data.Set 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 (here the libraries providing ISet and IMap are imported as IS and IM, respectively):

```haskell
type Store s = IM.IMap Addr s (IS.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 a function that amounts to 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:

```haskell
summarize :: IS.ISet s (State s) ! Par d s (Store s)
summarize states = do
  storeFin ← newEmptyMap
  void $ IS.forEach states $ λ (State _ _ store_n _) → do
    void $ IM.forEach store_n $ λ key val → do
      void $ IS.forEach val $ λ elem → do
        IM.modify storeFin key newEmptySet $ λ st → do
          IS.insert elem st
        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 operations 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 to LVish 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 (called in the last line of Listing 4.12). In keeping with the purely functional program from which it was ported, `next` creates a fresh store to extend with new bindings as we take each step through the state space graph:

```haskell
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:

```haskell
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), is 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 [29].

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 [44].

---

DRAFT: March 14, 2015
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. [19]. 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 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 4.1 shows the parallel speedup results of our experiments on a twelve-core machine.\footnote{Intel Xeon 5660; full machine details available at https://portal.futuregrid.org/hardware/delta.} (We used $k = 2$ for the benchmarks in this section.) The lines labeled “blur” and “blur/lockfree” show the parallel 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 lock-free shared data structure results in a better parallel speedup; rather, the interesting thing about these results is that despite its determinism guarantee,
Figure 4.1. 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 (not shown) 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.

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. Indeed, part of the benefit of LVish is that it can 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 Haskell programs access to efficient in-place array updates.
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 [45] is a medium-sized (3500-line) bioinformatics program implemented in Haskell\(^\text{16}\) 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 \([57]\), which is used by a software package of the same name.\(^\text{17}\) The HashRF software package is written in C++ and is about 2-3× as fast as PhyBin, which also implements the HashRF algorithm. Both packages are dozens or hundreds of times faster than the more widely-used software that computes RF distance matrices, such as PHYLIP\(^\text{18}\) [22] and DendroPy\(^\text{19}\) [56]. 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.

\(^{16}\)Available at http://hackage.haskell.org/package/phybin.
\(^{17}\)Available at https://code.google.com/p/hashrf/.
\(^{19}\)Available at http://pythonhosted.org/DendroPy/.
Algorithm 1 Pseudocode of the HashRF algorithm for computing a tree edit distance matrix. alltrees, splitsmap and distancematrix are global variables, defined elsewhere. alltrees is the set of trees, represented as sets of bipartitions; splitsmap maps bipartitions to sets of trees in which they occur. In the second phase, the comparison of $t_1$ and $t_2$ 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.

▷ First phase: populate splits map.

for each $t \in$ alltrees do
  for each bip $\in$ $t$ do
    ▷ Add $t$ to set of trees pointed at by splitsmap[bip],
    ▷ adding a key for bip to splitsmap if necessary.
    insert($t$, splitsmap[bip])
  end for
end for

▷ Second phase: populate distance matrix.

▷ values() returns a list of all the values in a dictionary.

for each treeset $\in$ values(splitsmap) do
  for each $t_1 \in$ alltrees do
    for each $t_2 \in$ alltrees do
      if $t_1 \in$ treeset XOR $t_2 \in$ treeset then
        increment(distancematrix[$t_1$, $t_2$])
      end if
    end for
  end for
end for

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.

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 dictionary, 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.

Algorithm 1 is a psuedocode version of the HashRF algorithm. The second phase of the algorithm is still $O(N^2)$, but it only needs to read from the much smaller treeset during this phase. All loops in Algorithm 1 are potentially parallel.

### 4.6.2. Parallelizing the HashRF algorithm with LVish.

In the original PhyBin source code, the type of the splits map is:

```haskell
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 C++ HashRF implementation uses a mutable hash table. Yet in both cases, these structures grow monotonically during execution, making the algorithm a good candidate for parallelization with LVish. The splits map created during the first phase of the algorithm is a map of sets, which can be directly replaced by their LVar counterparts, and the distance matrix created in the second phase can be represented as a vector of monotonically increasing counters.

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.\(^{20}\) Tables 4.1 and 4.2 show 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. Table 4.1 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 the two alternatives.

\(^{20}\)Git commit range: [https://github.com/rrnewton/PhyBin/compare/5cbf7d26c07a...6a05cfab490a7a](https://github.com/rrnewton/PhyBin/compare/5cbf7d26c07a...6a05cfab490a7a).
4. THE LVISH LIBRARY: INTERFACE, IMPLEMENTATION, AND EVALUATION

Table 4.1. PhyBin performance comparison with DendroPy and PHYLIP.

<table>
<thead>
<tr>
<th>Trees</th>
<th>Species</th>
<th>DendroPy</th>
<th>PHYLIP</th>
<th>PhyBin</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>150</td>
<td>22.1s</td>
<td>12.8s</td>
<td><strong>0.269s</strong></td>
</tr>
</tbody>
</table>

Then, to compare PhyBin with HashRF, we used a set of 1000 trees with 150 leaves each. Table 4.2 shows the results. HashRF took about 1.7 seconds to process the 1000 trees, but since it is a single-threaded program, 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. Therefore LVish gives us a parallel speedup of about $3.35 \times$ on 8 cores.

This is exactly the sort of situation in which we would like to use LVish—to achieve modest speedups for modest effort, in programs with complex data structures (and high allocation rates), and without changing the determinism guarantee of the original functional code.

Table 4.2. PhyBin performance comparison with HashRF.

<table>
<thead>
<tr>
<th>Trees</th>
<th>Species</th>
<th>HashRF</th>
<th>PhyBin</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>150</td>
<td><strong>1.7s</strong></td>
<td>1.4s</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>1 core</th>
<th>2 cores</th>
<th>4 cores</th>
<th>8 cores</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000 150</td>
<td>4.7s</td>
<td>3.0s</td>
<td>1.9s</td>
<td><strong>1.4s</strong></td>
</tr>
</tbody>
</table>
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 [26, 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 [61], 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, an 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. Fortunately, we need not implement them in an ad-hoc way. Shapiro et al.’s convergent replicated data types (CvRDTs) [54, 53] 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 [59], 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 [62].

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.

\[\text{Indeed, as the developers of Dynamo have noted [17], Amazon’s shopping cart presents an anomaly whereby removed items may re-appear in the cart!}\]
5. DETERMINISTIC THRESHOLD QUERIES OF DISTRIBUTED DATA STRUCTURES

That is, a strongly consistent query of a distributed data structure, if it returns, will return a result that is a \textit{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 \textit{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 \textit{threshold queries}. After reviewing the fundamentals of CvRDTs in Section \ref{sec:background}, I introduce CvRDTs extended with threshold queries (Section \ref{sec:threshold}) and prove that threshold queries in our extended model are strongly consistent queries (Section \ref{sec:threshold-consistency}). 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.

\section{Background: CvRDTs and eventual consistency

Shapiro \textit{et al.} \cite{54, 53} define an \textit{eventually consistent} object as one that meets three conditions. One of these conditions is the property of \textit{convergence}: all correct replicas of an object at which the same updates have been delivered eventually have equivalent state. The other two conditions are \textit{eventual delivery}, meaning that all replicas receive all update messages, and \textit{termination}, meaning that all method executions terminate (we discuss methods in more detail below).

Shapiro \textit{et al.} further define a \textit{strongly eventually consistent} (SEC) object as one that is eventually consistent and, in addition to being merely convergent, is \textit{strongly convergent}, meaning that correct replicas
at which the same updates have been delivered have equivalent state.\footnote{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.} A conflict-free replicated data type (CRDT), then, is a data type (\textit{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 
\textit{convergent}\footnote{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.}; or \textit{operation-based} (or “op-based”), also known as \textit{commutative}. CRDTs specified in the state-based style are called \textit{convergent replicated data types}, abbreviated CvRDTs, while those specified in the op-based style are called \textit{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 \textit{et al.} show, CmRDTs can emulate CvRDTs and vice versa.

\subsection*{5.1.1. State-based objects.} The Shapiro \textit{et al.} model specifies a \textit{state-based object} as a tuple $\langle S, s^0, q, u, m \rangle$, where $S$ is a set of states, $s^0$ is the initial state, $q$ is the \textit{query method}, $u$ is the \textit{update method}, and $m$ is the \textit{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 \textit{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
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 [54].

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 $S = \{S_a, S_b, \ldots\}$ 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 $S_a, S_b \in S$, 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 \( k \)th method execution at replica \( i \) is written \( f^k_i(a) \), where \( k \) is \( \geq 1 \) 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 \( k \)th method execution at \( i \) is \( s^k_i \). 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 \( k \)th 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: \( c^k_i = c^{k-1}_i \).
- an update \( u^k_i(a) \), then the causal history at replica \( i \) after execution \( k \) is \( c^k_i = c^{k-1}_i \cup u^k_i(a) \).
- a merge \( m^k_i(s^{k'}_{i'}) \), then the causal history at replica \( i \) after execution \( k \) is the union of the local and remote histories: \( c^k_i = c^{k-1}_i \cup c^{k'}_{i'} \).

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 \( S \) as its argument, and has the following semantics: let \( t^{k+1}_i(S) \) 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 \( S_a \in S \), the condition \( s_a \leq s^k_i \) is met, \( t^{k+1}_i(S) \) returns the set of activation states \( S_a \). Otherwise, \( t^{k+1}_i(S) \) 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: $\forall i, j : c_i = c_j \Rightarrow s_i \equiv s_j$.

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 [54]. 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 lub 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.20 for the complete proof. \( \square \)

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 2.4.2 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 \texttt{get} in the presence of conflicting writes is not observable: such a program would always have \texttt{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 [49], only allowing single assignments to shared references [60, 3, 11], allowing sharing only by a limited form of message passing [32], ensuring that concurrent accesses to shared state are disjoint [8], resolving conflicting updates after the fact [36], 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 [32], 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 \(\lambda_{\text{LVar}}\) with a lattice with one “empty” state and multiple “full” states (where \(\forall i. \text{empty} < \text{full}_i\)) 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. [50] 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.
6. RELATED WORK

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 [36]. 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 lub 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 lub).

Although versioned variables in CR could model lattice-based data structures—if they used the lub operation 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

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 [54, 53] 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 LVars effects.
6. RELATED WORK

6.6. Frame properties and separation logics

In Section 2.5.5, we saw that the Independence lemma, Lemma 2.5, expresses a frame property reminiscent of the following frame rule from separation logic and concurrent separation logic [48, 52, 47]:

\[
\frac{\{p\} C \{q\}}{\{p * r\} C \{q * r\}}
\]

Recall that the separating conjunction connective \(\ast\) says that the assertions it combines can be satisfied in a non-overlapping manner; for instance, \(p \ast r\) is satisfied by a heap if the heap can be split into two non-overlapping parts satisfying \(p\) and \(r\), respectively. However, sometimes we do in fact want to allow some amount of “physical” overlap between resources, while retaining “logical” or “fictional” separation. In fact, the Independence lemma, since it replaces the separating conjunction with the lub operation, allows overlap between the original store and the “frame” store \(S''\); indeed, the point of LVars is that total disjointness is unnecessary, since updates commute. Jensen and Birkedal’s recent work on fictional separation logic [31] explores the notion of fictional separation in detail, generalizing traditional separation logic to allow much more sophisticated kinds of sharing.

Even more recently, Dinsdale-Young et al. [18] introduced the “Views” framework, which brings the notion of fictional separation to a concurrent setting. The Views framework is a metatheory of concurrent reasoning principles that generalizes a variety of concurrent program logics and type systems, including concurrent separation logic. It provides a generalized frame rule, which is parameterized by a function \(f\) that is applied to the pre- and post-conditions in the conclusion of the rule:

\[
\frac{\{p\} C \{q\}}{\{f(p)\} C \{f(q)\}}
\]

In this formulation of the rule, the “frame” is an abstract piece of knowledge that is not violated by the execution of \(C\). The Generalized Independence lemma (Lemma 3.7) that I describe in Section 3.3.5,
which extends the Independence lemma to handle arbitrary update operations, is reminiscent of this generalized frame rule.
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 unifying abstraction 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 $\lambda_{\text{Var}}$ and $\lambda_{\text{LVish}}$ 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 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 $\forall i. \text{empty} < \text{full}$). 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, we have identified a new class of quasi-deterministic programs and developed a programming model that supports quasi-determinism by construction. A quasi-deterministic model allows programs that perform freezing and are deterministic modulo write-after-freeze 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 4.2.5, 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 $\lambda_{LVish}$. Moreover, even for the parts of the LVish library that are fully accounted for in the model, we do not have proof that the 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/39233703496871424](https://twitter.com/neil_conway/status/39233703496871424)).
7. SUMMARY AND FUTURE WORK

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 individually 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) [54, 53], 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 lub 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 LV 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


[34] N. R. Krishnaswami. Higher-order reactive programming without spacetime leaks. In ICFP, 2013. 56


APPENDIX A

Proofs

A.1. Proof of Lemma 2.1

Proof. Consider an arbitrary permutation $\pi$. For part 1, we have to show that if $\sigma \rightarrow \sigma'$ then $\pi(\sigma) \rightarrow \pi(\sigma')$, and that if $\pi(\sigma) \rightarrow \pi(\sigma')$ then $\sigma \rightarrow \sigma'$.

For the forward direction of part 1, suppose $\sigma \rightarrow \sigma'$. We have to show that $\pi(\sigma) \rightarrow \pi(\sigma')$. We proceed by cases on the rule by which $\sigma$ steps to $\sigma'$.

- **Case E-Beta:** $\sigma = \langle S; (\lambda x. e) \, v \rangle$, and $\sigma' = \langle S; e[x := v] \rangle$.
  
  To show: $\pi(\langle S; (\lambda x. e) \, v \rangle) \rightarrow \pi(\langle S; e[x := v] \rangle)$.
  
  By Definitions 2.9 and 2.7, $\pi(\sigma) = \langle \pi(S); (\lambda x. \pi(e)) \, \pi(v) \rangle$.
  
  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:** $\sigma = \langle S; \text{new} \rangle$, and $\sigma' = \langle S[l \rightarrow \bot]; l \rangle$.
  
  To show: $\pi(\langle S; \text{new} \rangle) \rightarrow \pi(\langle S[l \rightarrow \bot]; l \rangle)$.
  
  By Definitions 2.9 and 2.7, $\pi(\sigma) = \langle \pi(S); \text{new} \rangle$.
  
  By E-New, $\langle \pi(S); \text{new} \rangle$ steps to $\langle \pi(S)[l' \rightarrow \bot]; l' \rangle$, where $l' \notin \text{dom}(\pi(S))$.
  
  It remains to show that $\langle \pi(S)[l' \rightarrow \bot]; l' \rangle$ is equal to $\pi(\langle S[l \rightarrow \bot]; l \rangle)$.
  
  By Definition 2.9, $\pi(\langle S[l \rightarrow \bot]; l \rangle)$ is equal to $\langle \pi(S[l \rightarrow \bot]); \pi(l) \rangle$, which is equal to $\langle \pi(S)[\pi(l) \rightarrow \bot]; \pi(l) \rangle$.
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 $l \notin dom(S)$, it follows that $\pi(l) \notin \pi(dom(S))$. Therefore, in $\langle (\pi(S))[l' \mapsto \bot]; l' \rangle$, we can $\alpha$-rename $l'$ to $\pi(l)$, and so the two configurations are equal and the case is satisfied.

- **Case E-Put:** $\sigma = \langle S; \text{put } l \ d_2 \rangle$, and $\sigma' = \langle S[l \mapsto d_1 \cup d_2]; \ O \rangle$.

To show: $\pi(\langle S; \text{put } l \ d_2 \rangle) \longmapsto \pi(\langle S[l \mapsto d_1 \cup d_2]; \ O \rangle)$. By Definitions 2.9 and 2.7, $\pi(\sigma) = \langle \pi(S); \text{put } \pi(l) \ d_2 \rangle$.

By E-Put, $\langle \pi(S); \text{put } \pi(l) \ d_2 \rangle$ steps to $\langle (\pi(S))[\pi(l) \mapsto d_1 \cup d_2]; \ O \rangle$, since $S(l) = (\pi(S))(\pi(l)) = 1$.

It remains to show that $\langle (\pi(S))[\pi(l) \mapsto d_1 \cup d_2]; \ O \rangle$ is equal to $\pi(\langle S[l \mapsto d_1 \cup d_2]; \ O \rangle)$. By Definitions 2.9 and 2.7, $\pi(\langle S[l \mapsto d_1 \cup d_2]; \ O \rangle)$ is equal to $\langle (\pi(S))[\pi(l) \mapsto d_1 \cup d_2]; \ O \rangle$, and so the two configurations are equal and the case is satisfied.

- **Case E-Put-Err:** $\sigma = \langle S; \text{put } l \ d_2 \rangle$, and $\sigma' = \textbf{error}$.

To show: $\pi(\langle S; \text{put } l \ d_2 \rangle) \longmapsto \pi(\textbf{error})$. By Definitions 2.9 and 2.7, $\pi(\sigma) = \langle \pi(S); \text{put } \pi(l) \ d_2 \rangle$.

By E-Put-Err, $\langle \pi(S); \text{put } \pi(l) \ d_2 \rangle$ steps to $\textbf{error}$, since $S(l) = (\pi(S))(\pi(l)) = 1$.

Since $\pi(\textbf{error}) = \textbf{error}$ by Definition 2.9, the case is complete.

- **Case E-Get:** $\sigma = \langle S; \text{get } l \ T \rangle$, and $\sigma' = \langle S; d_2 \rangle$.

To show: $\pi(\langle S; \text{get } l \ T \rangle) \longmapsto \pi(\langle S; d_2 \rangle)$. By Definitions 2.9 and 2.7, $\pi(\sigma) = \langle \pi(S); \text{get } \pi(l) \ T \rangle$.

By E-Get, $\langle \pi(S); \text{get } \pi(l) \ T \rangle$ steps to $\langle \pi(S); d_2 \rangle$, since $S(l) = (\pi(S))(\pi(l)) = 1$.

By Definitions 2.9 and 2.7, $\pi(\langle S; d_2 \rangle) \longmapsto \pi(\langle S; d_2 \rangle)$. Therefore the case is complete.

For the reverse direction of part 1, 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 $\sigma$ and $\sigma'$ and permutations $\pi$, if $\sigma \longmapsto \sigma'$ then $\pi(\sigma) \longmapsto \pi(\sigma')$. Hence since $\pi(\sigma) \longmapsto \pi(\sigma')$, and since $\pi^{-1}$ is also a permutation,
we have that $\pi^{-1}(\pi(\sigma)) \leftrightarrow \pi^{-1}(\pi(\sigma'))$. Since $\pi^{-1}(\pi(l)) = l$ for every $l \in \text{Loc}$, and that property lifts to configurations as well, we have that $\sigma \leftrightarrow \sigma'$.

LK: Is the above enough of a proof?

For the forward direction of part 2, suppose $\sigma \leftrightarrow \sigma'$. We have to show that $\pi(\sigma) \leftrightarrow \pi(\sigma')$.

By inspection of the operational semantics, $\sigma$ must be of the form $\langle S; \ E[e] \rangle$, and $\sigma'$ must be of the form $\langle S'; \ E[e'] \rangle$. Hence we have to show that $\pi(\langle S; \ E[e] \rangle) \leftrightarrow \pi(\langle S'; \ E[e'] \rangle)$.

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 $\langle \pi(S); \ \langle \pi(E) [\pi(e)] \rangle \rangle$ and $\langle \pi(S'); \ \pi(E[e']) \rangle$ is equal to $\langle \pi(S'); \ \langle \pi(E) [\pi(e')] \rangle \rangle$.

So we have to show that $\langle \pi(S); \ \langle \pi(E) [\pi(e)] \rangle \rangle \leftrightarrow \langle \pi(S'); \ \langle \pi(E) [\pi(e')] \rangle \rangle$.

From the premise of E-Eval-Ctxt, $\langle S; \ e \rangle \leftrightarrow \langle S'; \ e' \rangle$. Hence, by part 1, $\pi(\langle S; \ e \rangle) \leftrightarrow \pi(\langle S'; \ e' \rangle)$.

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 $\langle \pi(S); \ \pi(e) \rangle \leftrightarrow \langle \pi(S'); \ \pi(e') \rangle$. Therefore, by E-Eval-Ctxt, $\langle \pi(S); \ E[\pi(e)] \rangle \leftrightarrow \langle \pi(S'); \ E[\pi(e')] \rangle$ for all evaluation contexts $E$.

In particular, it is true that $\langle \pi(S); \ \langle \pi(E) [\pi(e)] \rangle \rangle \leftrightarrow \langle \pi(S'); \ \langle \pi(E) [\pi(e')] \rangle \rangle$, as we were required to show.

For the reverse direction of part 2, suppose $\pi(\sigma) \leftrightarrow \pi(\sigma')$. We have to show that $\sigma \leftrightarrow \sigma'$.

We know from the forward direction of the proof that for all configurations $\sigma$ and $\sigma'$ and permutations $\pi$, if $\sigma \leftrightarrow \sigma'$ then $\pi(\sigma) \leftrightarrow \pi(\sigma')$. Hence since $\pi(\sigma) \leftrightarrow \pi(\sigma')$, and since $\pi^{-1}$ is also a permutation, we have that $\pi^{-1}(\pi(\sigma)) \leftrightarrow \pi^{-1}(\pi(\sigma'))$. Since $\pi^{-1}(\pi(l)) = l$ for every $l \in \text{Loc}$, and that property lifts to configurations as well, we have that $\sigma \leftrightarrow \sigma'$.

LK: Is the above enough of a proof?

□
A.2. Proof of Lemma 2.2

Proof. Suppose $\sigma \rightarrow \sigma'$ and $\sigma \rightarrow \sigma''$. We have to show that there is a permutation $\pi$ such that $\sigma' = \pi(\sigma'')$. The proof is by cases on the rule by which $\sigma$ steps to $\sigma'$.

- **Case E-Beta:**
  
  Given: $\langle S; (\lambda x. e) \nu \rangle \rightarrow \langle S; e[x := \nu] \rangle$, and $\langle S; (\lambda x. e) \nu \rangle \rightarrow \sigma''$.
  
  To show: There exists a $\pi$ such that $\langle S; e[x := \nu] \rangle = \pi(\sigma'')$.
  
  By inspection of the operational semantics, the only reduction rule by which $\langle S; (\lambda x. e) \nu \rangle$ can step is E-Beta. Hence $\sigma'' = \langle S; e[x := \nu] \rangle$, and the case is satisfied by choosing $\pi$ to be the identity function.

- **Case E-New:**
  
  Given: $\langle S; \text{new} \rangle \rightarrow \langle S[l \mapsto \bot]; l \rangle$, and $\langle S; \text{new} \rangle \rightarrow \sigma''$.
  
  To show: There exists a $\pi$ such that $\langle S[l \mapsto \bot]; l \rangle = \pi(\sigma'')$.
  
  By inspection of the operational semantics, the only reduction rule by which $\langle S; \text{new} \rangle$ 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 $\text{dom}(S)$, the case is satisfied by choosing $\pi$ to be the permutation that maps $l'$ to $l$ and is the identity on every other element of Loc.

- **Case E-Put:**
  
  Given: $\langle S; \text{put } l \ 	ext{d} \ 	ext{d}_2 \rangle \rightarrow \langle S[l \mapsto d_1 \sqcup d_2]; \bot \rangle$, and $\langle S; \text{put } l \ 	ext{d}_2 \rangle \rightarrow \sigma''$.
  
  To show: There exists a $\pi$ such that $\langle S[l \mapsto d_1 \sqcup d_2]; \bot \rangle = \pi(\sigma'')$.
  
  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 $\langle S; \text{put } l \ 	ext{d}_2 \rangle$ can step is E-Put. Hence $\sigma'' = \langle S[l \mapsto d_1 \sqcup d_2]; \bot \rangle$, and the case is satisfied by choosing $\pi$ to be the identity function.

- **Case E-Put-Err:**
  
  Given: $\langle S; \text{put } l \ 	ext{d}_2 \rangle \rightarrow \text{error}$, and $\langle S; \text{put } l \ 	ext{d}_2 \rangle \rightarrow \sigma''$.
  
  To show: There exists a $\pi$ such that $\text{error} = \pi(\sigma'')$.

DRAFT: March 14, 2015 154
A. PROOFS

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 \( \langle S; \text{put} \ l \ d_2 \rangle \) can step is E-Put-Err. Hence \( \sigma'' = \text{error} \), and the case is satisfied by choosing \( \pi \) to be the identity function.

- **Case E-Get:**
  
  Given: \( \langle S; \text{get} \ l \ T \rangle \rightarrow \langle S; d_2 \rangle \), and \( \langle S; \text{get} \ l \ T \rangle \rightarrow \sigma'' \).
  
  To show: There exists a \( \pi \) such that \( \langle S; d_2 \rangle = \pi(\sigma'') \).
  
  By inspection of the operational semantics, the only reduction rule by which \( \langle S; \text{get} \ l \ T \rangle \) can step is E-Get. Hence \( \sigma'' = \langle S; d_2 \rangle \), and the case is satisfied by choosing \( \pi \) to be the identity function.

\[ \square \]

A.3. Proof of Lemma 2.3

Proof. **TODO: Prove this.**

\[ \square \]

A.4. Proof of Lemma 2.4

Proof. Suppose \( \langle S; e \rangle \rightarrow \langle S'; e' \rangle \). 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: \( \langle S; \text{new} \rangle \rightarrow \langle S[l \mapsto \bot]; l \rangle \).
  
  To show: \( S \sqsubseteq_S S[l \mapsto \bot] \).
  
  By Definition 2.2, we have to show that \( \text{dom}(S) \subseteq \text{dom}(S[l \mapsto \bot]) \) and that for all \( l' \in \text{dom}(S) \), \( S(l') \subseteq (S[l \mapsto \bot])(l') \).
  
  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 \( \text{dom}(S) \subseteq \text{dom}(S[l \mapsto \bot]) \).
From the side condition of E-New, \( l \notin \text{dom}(S) \). 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 \( l' \in \text{dom}(S) \), \( S(l') \subseteq (S[l \mapsto \bot])(l') \).

Therefore \( S \subseteq S[l \mapsto \bot] \), as required.

**Case E-Put:**

Given: \( \langle S; \text{put } l \; d_2 \rangle \leftarrow \langle S[l \mapsto d_1 \sqcup d_2]; \; () \rangle \).

To show: \( S \subseteq S[l \mapsto d_1 \sqcup d_2] \).

By Definition 2.2, we have to show that \( \text{dom}(S) \subseteq \text{dom}(S[l \mapsto d_1 \sqcup d_2]) \) and that for all \( l' \in \text{dom}(S) \), \( S(l') \subseteq (S[l \mapsto d_1 \sqcup d_2])(l') \).

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 \( \text{dom}(S) \subseteq \text{dom}(S[l \mapsto d_1 \sqcup d_2]) \).

From the premises of E-Put, \( S(l) = d_1 \). Therefore \( l \in \text{dom}(S) \).

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 \( \sqcup \), \( d_1 \sqcup (d_1 \sqcup d_2) \). \( S[l \mapsto d_1 \sqcup d_2] \) does not update any other bindings in \( S \), hence, for all \( l' \in \text{dom}(S) \), \( S(l') \subseteq (S[l \mapsto d_1 \sqcup d_2])(l') \).

Hence \( S \subseteq S[l \mapsto d_1 \sqcup d_2] \), as required.

**Case E-Put-Err:**

Given: \( \langle S; \text{put } l \; d_2 \rangle \leftarrow \text{error} \).

By the definition of \text{error}, \text{error} is equal to \( \langle \top_S; \; e \rangle \) for all \( e \).

To show: \( S \subseteq \top_S \).

Immediate by the definition of \( \subseteq_S \).

**Case E-Get:**

Immediate by the definition of \( \subseteq_S \), since \( S \) does not change.
A. PROOFS

A.5. Proof of Lemma 2.5

Proof. Suppose \( \langle S; e \rangle \leftarrow \langle S'; e' \rangle \), where \( \langle S'; e' \rangle \neq \text{error} \). Consider arbitrary \( S'' \) such that \( S'' \) is non-conflicting with \( \langle S; e \rangle \leftarrow \langle S'; e' \rangle \) and \( S' \sqcup_S S'' \neq \top_S \). We are required to show that \( \langle S \sqcup_S S''; e \rangle \leftarrow \langle S' \sqcup_S S''; e' \rangle \).

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 \( \langle S'; e' \rangle \neq \text{error} \), we do not need to consider the E-Put-Err rule.

- **Case E-Beta:**
  
  Given: \( \langle S; (\lambda x. e) \, v \rangle \leftarrow \langle S; e \, [x := v] \rangle \).
  
  To show: \( \langle S \sqcup_S S''; (\lambda x. e) \, v \rangle \leftarrow \langle S \sqcup_S S''; e \, [x := v] \rangle \).
  
  Immediate by E-Beta.

- **Case E-New:**
  
  Given: \( \langle S; \text{new} \rangle \leftarrow \langle S[l \mapsto \bot]; l \rangle \).
  
  To show: \( \langle S \sqcup_S S''; \text{new} \rangle \leftarrow \langle (S[l \mapsto \bot]) \sqcup_S S''; l \rangle \).
  
  By E-New, we have that \( \langle S \sqcup_S S''; \text{new} \rangle \leftarrow \langle (S[l \mapsto \bot]) \sqcup_S S''; l \rangle \), where \( l' \notin \text{dom}(S \sqcup_S S'') \).
  
  By assumption, \( S'' \) is non-conflicting with \( \langle S; \text{new} \rangle \leftarrow \langle S[l \mapsto \bot]; l \rangle \).
  
  Therefore \( l \notin \text{dom}(S'') \).
  
  From the side condition of E-New, \( l \notin \text{dom}(S) \).
  
  Therefore \( l \notin \text{dom}(S \sqcup_S S'') \).
  
  Therefore, in \( \langle (S \sqcup_S S'')[l' \mapsto \bot]; l' \rangle \), we can \( \alpha \)-rename \( l' \) to \( l \), resulting in \( \langle (S \sqcup_S S'')[l \mapsto \bot]; l \rangle \).
  
  Therefore \( \langle S \sqcup_S S''; \text{new} \rangle \leftarrow \langle (S \sqcup_S S'')[l \mapsto \bot]; l \rangle \).
Note that:

\[
(S \cup S')[l \mapsto \bot] = S[l \mapsto \bot] \cup S' [l \mapsto \bot] \\
= S \cup [l \mapsto \bot] \cup S' [l \mapsto \bot] \\
= S \cup [l \mapsto \bot] \cup S'' \\
= S[l \mapsto \bot] \cup S''.
\]

Therefore \(\langle S \cup S'' \rangle; \text{new}\) \(\longrightarrow \langle S[l \mapsto \bot] \cup S''; l \rangle\), as we were required to show.

- **Case E-Put:**

  **Given:** \(\langle S; \text{put } l \ d_2 \rangle \longrightarrow \langle S[l \mapsto d_2]; \emptyset \rangle\).

  **To show:** \(\langle S \cup S'' ; \text{put } l \ d_2 \rangle \longrightarrow \langle S[l \mapsto d_2] \cup S'' ; \emptyset \rangle\).

  We will first show that

  \(\langle S \cup S'' ; \text{put } l \ d_2 \rangle \longrightarrow \langle (S \cup S'') [l \mapsto d_2]; \emptyset \rangle\)

  and then show why this is sufficient.

  We proceed by cases on \(l\):

  - \(l \notin \text{dom}(S'')\):
    
    By assumption, \(S[l \mapsto d_2] \cup S'' \neq \top_S\).
    
    By Lemma 2.4, \(S \subseteq_S S[l \mapsto d_2]\).
    
    Hence \(S \cup_S S'' \neq \top_S\).
    
    Therefore, by Definition 2.3, \((S \cup_S S'')(l) = S(l)\).
    
    From the premises of E-Put, \(S(l) = d_1\).
    
    Hence \((S \cup_S S'')(l) = d_1\).
    
    From the premises of E-Put, \(d_2 = d_1 \cup d_2\) and \(d_2 \neq \top\).
    
    Therefore, by E-Put, we have: \(\langle S \cup_S S'' ; \text{put } l \ d_2 \rangle \longrightarrow \langle (S \cup_S S'')[l \mapsto d_2]; \emptyset \rangle\).
    
  - \(l \in \text{dom}(S'')\):
    
    By assumption, \(S[l \mapsto d_2] \cup S'' \neq \top_S\).
By Lemma 2.4, $S \subseteq S[l \mapsto d_2]$.

Hence $S \cup S'' \neq \top_S$.

Therefore $(S \cup S'')(l) = S(l) \cup S''(l)$.

From the premises of E-Put, $S(l) = d_1$.

Hence $(S \cup S'')(l) = d_1'$, 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, $S[l \mapsto d_2] \cup S'' \neq \top_S$.

Therefore, by Definition 2.3, $d_2 \cup S''(l) \neq \top$.

Note that:

$$
\top \neq d_2 \cup S''(l) \\
= d_1 \sqcup d_2 \cup S''(l) \\
= S(l) \sqcup d_2 \cup S''(l) \\
= S(l) \sqcup S''(l) \sqcup d_2 \\
= (S \cup_S S'')(l) \sqcup d_2 \\
= d_1' \sqcup d_2 \\
= d_2'.
$$

Hence $d_2' \neq \top$.

Hence $(S \cup S'')(l) = d_1'$ and $d_2' = d_1' \sqcup d_2$ and $d_2' \neq \top$.

Therefore, by E-Put we have: $\langle S \cup_S S''; \text{ put } l \mapsto d_2 \rangle \iff \langle (S \cup_S S'')[l \mapsto d_2]; \top \rangle$. 


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 \((S \sqcup S')[l \mapsto d''_2]\) and \((S \sqcup S')[l \mapsto d_2]\) bind \(l\) to the same value, then it will be obvious that they’re equal.

Note that:

\[(S \sqcup S')[l \mapsto d''_2](l) = (S \sqcup S')(l) \sqcup ([l \mapsto d''_2])(l)\]

\[= d'_1 \sqcup d''_2\]

\[= d'_1 \sqcup d'_1 \sqcup d_2\]

\[= d'_1 \sqcup d_2\]

and

\[(S \sqcup S')[l \mapsto d_2](l) = (S \sqcup S')(l) \sqcup ([l \mapsto d_2])(l)\]

\[= d'_1 \sqcup d_2\]

\[= d'_1 \sqcup d'_1 \sqcup d_2\]

\[= d'_1 \sqcup d_2\] (since \(d_1 \sqsubseteq d'_1\)).

Therefore \((S \sqcup S')[l \mapsto d''_2] = (S \sqcup S')[l \mapsto d_2]\).

Therefore, \(\langle S \sqcup S'; \text{put } l \mapsto d_2 \rangle \leftrightarrow \langle (S \sqcup S')[l \mapsto d_2]; \emptyset \rangle\).

Note that:

\[(S \sqcup S')[l \mapsto d_2] = S[l \mapsto d_2] \sqcup S'[l \mapsto d_2]\]

\[= S \sqcup S[l \mapsto d_2] \sqcup S'' \sqcup S[l \mapsto d_2]\]

\[= S \sqcup S[l \mapsto d_2] \sqcup S''\]

\[= S[l \mapsto d_2] \sqcup S''\].
A. PROOFS

Therefore $\langle S \sqcup S'; \text{put } l \ d_2 \rangle \rightarrow \langle S[l \mapsto d_2] \sqcup S''; \ 0 \rangle$, as we were required to show.

- **Case E-Get:**

  Given: $\langle S; \text{get } l \ T \rangle \rightarrow \langle S; d_2 \rangle$.

  To show: $\langle S \sqcup S''; \text{get } l \ T \rangle \rightarrow \langle S \sqcup S''; d_2 \rangle$.

  From the premises of E-Get, $S(l) = d_1$ and $\text{incomp}(T)$ and $d_2 \in T$ and $d_2 \sqsubseteq d_1$.

  By assumption, $S \sqcup S'' \neq \top_S$.

  Hence $(S \sqcup S'')(l) = d'_1$, where $d_1 \sqsubseteq d'_1$.

  By the transitivity of $\sqsubseteq$, $d_2 \sqsubseteq d'_1$.

  Hence, $(S \sqcup S'')(l) = d'_1$ and $\text{incomp}(T)$ and $d_2 \in T$ and $d_2 \sqsubseteq d'_1$.

  Therefore, by E-Get,

  $\langle S \sqcup S''; \text{get } l \ T \rangle \rightarrow \langle S \sqcup S''; d_2 \rangle$,

  as we were required to show.

\[\square\]

A.6. **Proof of Lemma 2.6**

Proof. Suppose $\langle S; e \rangle \rightarrow \langle S'; e' \rangle$, where $\langle S'; e' \rangle \neq \text{error}$. Consider arbitrary $S''$ such that $S''$ is non-conflicting with $\langle S; e \rangle \rightarrow \langle S'; e' \rangle$ and $S' \sqcup S'' = \top_S$. We are required to show that there exists $i \leq 1$ such that $\langle S \sqcup S'; e \rangle \rightarrow^i \text{error}$.

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 $\langle S'; e' \rangle \neq \text{error}$, we do not need to consider the E-Put-Err rule.

- **Case E-Beta:**

  Given: $\langle S; (\lambda x. e) \ b \rangle \rightarrow \langle S; e[x := v] \rangle$.

  To show: $\langle S \sqcup S''; (\lambda x. e) \ b \rangle \rightarrow^i \text{error}$, where $i \leq 1$.

  By assumption, $S \sqcup S'' = \top_S$.

  Hence, by the definition of $\text{error}$, $\langle S \sqcup S''; (\lambda x. e) \ b \rangle = \text{error}$. 
Hence $\langle S \uplus S''; (\lambda x. e) \nu \rangle \rightarrow^i \text{error}$, with $i = 0$.

- **Case E-New:**

  Given: $\langle S; \text{new} \rangle \rightarrow \langle S[l \mapsto \bot]; l \rangle$.

  To show: $\langle S \uplus S''; \text{new} \rangle \rightarrow^i \text{error}$, where $i \leq 1$.

  By E-New, $\langle S \uplus S''; \text{new} \rangle \rightarrow \langle (S \uplus S'')[l' \mapsto \bot]; l' \rangle$, where $l' \notin \text{dom}(S \uplus S'')$.

  By assumption, $S''$ is non-conflicting with $\langle S; \text{new} \rangle \rightarrow \langle S[l \mapsto \bot]; l \rangle$.

  Therefore $l \notin \text{dom}(S'')$.

  From the side condition of E-New, $l \notin \text{dom}(S \uplus S'')$.

  Therefore, in $\langle (S \uplus S'')[l' \mapsto \bot]; l' \rangle$, we can $\alpha$-rename $l'$ to $l$,

  resulting in $\langle (S \uplus S'')[l \mapsto \bot]; l \rangle$.

  Therefore $\langle S \uplus S''; \text{new} \rangle \rightarrow \langle (S \uplus S'')[l \mapsto \bot]; l \rangle$.

  By assumption, $S[l \mapsto \bot] \uplus S'' = \top_S$.

  Note that:

  \[
  \top_S = S[l \mapsto \bot] \uplus S''
  = S \uplus S[l \mapsto \bot] \uplus S''
  = S \uplus S'' \uplus S[l \mapsto \bot]
  = (S \uplus S'')[l \mapsto \bot]
  = (S \uplus S'')[l \mapsto \bot].
  \]

  Hence $\langle S \uplus S''; \text{new} \rangle \rightarrow \langle \top_S; l \rangle$.

  Hence, by the definition of error, $\langle S \uplus S''; \text{new} \rangle \rightarrow \text{error}$.

  Hence $\langle S \uplus S''; \text{new} \rangle \rightarrow^i \text{error}$, with $i = 1$.

- **Case E-Put:**

  Given: $\langle S; \text{put } l d_2 \rangle \rightarrow \langle S[l \mapsto d_2]; () \rangle$. 

DRAFT: March 14, 2015
A. PROOFS

To show: $\langle S \sqcup S' S''; \text{put } l d_2 \rangle \leftarrow^i \text{error}$, 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 \texttt{error}, $\langle S \sqcup_S S''; \text{put } l d_2 \rangle = \text{error}$.

  Hence $\langle S \sqcup_S S''; \text{put } l d_2 \rangle \leftarrow^i \text{error}$, 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 $d_1 \subseteq d'_1$.

  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 $l' \in \text{dom}(S[l \mapsto d_2]) \cap \text{dom}(S'')$ 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:

\[
\top = (S[l \mapsto d_2])(l) \sqcup S''(l)
= d_2 \sqcup S''(l)
= d_1 \sqcup d_2 \sqcup S''(l)
= S(l) \sqcup d_2 \sqcup S''(l)
= S(l) \sqcup S''(l) \sqcup d_2
= (S \sqcup S'')(l) \sqcup d_2
= d_1' \sqcup d_2.
\]

Hence \( d_1' \sqcup d_2 = \top \).

Hence, by E-Put-Err, \( \langle S \sqcup S'' \text{; put } l d_2 \rangle \longrightarrow \text{error} \).

Hence \( \langle S \sqcup S'' \text{; put } l d_2 \rangle \longrightarrow^i \text{error} \), with \( i = 1 \).

\begin{itemize}
  \item Case E-Get:
  \end{itemize}

Given: \( \langle S; \text{get } l T \rangle \longrightarrow \langle S; d_2 \rangle \).

To show: \( \langle S \sqcup S'' \text{; get } l T \rangle \longrightarrow^i \text{error} \), where \( i \leq 1 \).

By assumption, \( S \sqcup S'' = \top_S \).

Hence, by the definition of \textit{error}, \( \langle S \sqcup S'' \text{; get } l T \rangle = \text{error} \).

Hence \( \langle S \sqcup S'' \text{; get } l T \rangle \longrightarrow^i \text{error} \), with \( i = 0 \).

\[\square\]

A.7. Proof of Lemma 2.8

Proof. Suppose \( \sigma \longrightarrow \sigma_a \) and \( \sigma \longrightarrow \sigma_b \). We have to show that there exist \( \sigma_c, i, j, \pi \) such that \( \sigma_a \longrightarrow^i \sigma_c \) and \( \pi(\sigma_b) \longrightarrow^j \sigma_c \) and \( i \leq 1 \) and \( j \leq 1 \).
By inspection of the operational semantics, it must be the case that $\sigma$ steps to $\sigma_a$ 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 $\sigma$ steps to $\sigma_b$ 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.

First, consider the possibility that $E_a = E_b$ (and $e_{a_1} = e_{b_1}$). Since $\langle S; E_a[e_{a_1}] \rangle \hookrightarrow \langle S_a; E_a[e_{a_2}] \rangle$ by E-Eval-Ctxt and $\langle S; E_b[e_{b_1}] \rangle \hookrightarrow \langle S_b; E_b[e_{b_2}] \rangle$ by E-Eval-Ctxt, we have from the premise of E-Eval-Ctxt that $\langle S; e_{a_1} \rangle \hookrightarrow \langle S_a; e_{a_2} \rangle$ and $\langle S; e_{b_1} \rangle \hookrightarrow \langle S_b; e_{b_2} \rangle$. But then, since $e_{a_1} = e_{b_1}$, by Internal Determinism (Lemma 2.2) there is a permutation $\pi'$ such that $\langle S_a; e_{a_2} \rangle = \pi'((S_b; e_{b_2}))$. Then we can satisfy the proof by choosing $\sigma_c = \langle S_a; e_{a_2} \rangle$ and $i = 0$ and $j = 0$ and $\pi = \pi'$.

The rest of this proof deals with the more interesting case in which $E_a \neq E_b$ (and $e_{a_1} \neq e_{b_1}$). Since $\langle S; E_a[e_{a_1}] \rangle \hookrightarrow \langle S_a; E_a[e_{a_2}] \rangle$ and $\langle S; E_b[e_{b_1}] \rangle \hookrightarrow \langle S_b; E_b[e_{b_2}] \rangle$ and $E_a[e_{a_1}] = E_b[e_{b_1}]$, where $E_a \neq E_b$, 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 $\sigma_c = \text{error}$. In most cases, however, our approach will be to show that there exist $S'$, $i$, $j$, $\pi$ such that:

- $\langle S_a; E_a[e_{a_2}] \rangle \hookrightarrow^i S'$, $\langle S_a; E'_a[e_{a_2}] \rangle$, and
- $\pi((S_b; E_b[e_{b_2}])) \hookrightarrow^j S'$, $\langle S'_a; E'_a[e_{a_2}] \rangle$.

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:

- $\langle S_a; E'_b[e_{b_1}] \rangle \hookrightarrow^i S'$, $\langle S'_b; E'_b[e_{b_2}] \rangle$, and
We proceed by case analysis on the rule by which
Case E-Beta: We have
Case E-Put: We have
Case E-New: We have
(Independence), we have that
From the premise of E-Eval-Ctxt, we have that

(1) Case E-Beta: We have $S_a = S$.

We proceed by case analysis on the rule by which $\langle S; e_{a_1} \rangle \rightarrow \langle S_a; e_{a_2} \rangle$ steps to $\langle S_a; e_{a_2} \rangle$.

(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:

- $\langle S; E'_b[e_{b_1}] \rangle \rightarrow \langle S_a; E'_b[e_{b_2}] \rangle$, and
- $\langle S; E'_a[e_{a_1}] \rangle \rightarrow \langle S_b; E'_a[e_{a_2}] \rangle$,

both of which follow immediately from $\langle S; e_{a_1} \rangle \rightarrow \langle S_a; e_{a_2} \rangle$ and $\langle S; e_{b_1} \rangle \rightarrow \langle S_b; e_{b_2} \rangle$ 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:

- $\langle S; E'_b[e_{b_1}] \rangle \rightarrow \langle S_b; E'_b[e_{b_2}] \rangle$, and
- $\langle S_b; E'_a[e_{a_1}] \rangle \rightarrow \langle S_b; E'_a[e_{a_2}] \rangle$.

The first of these follows immediately from $\langle S; e_{b_1} \rangle \rightarrow \langle S_b; e_{b_2} \rangle$ 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 $\langle S; e_{a_1} \rangle \rightarrow \langle S_a; e_{a_2} \rangle$, we know that $[l \mapsto \bot]$ is non-conflicting with it, and we know that $S_a \sqcup_S [l \mapsto \bot] \neq T_S$ since $S_a$ is just $S$ and $S \sqcup_S [l \mapsto \bot]$ cannot be $T_S$. Therefore, by Lemma 2.5 (Independence), we have that $\langle S \sqcup_S [l \mapsto \bot]; e_{a_1} \rangle \rightarrow \langle S_a \sqcup_S [l \mapsto \bot]; e_{a_2} \rangle$. Hence $\langle S_b; e_{a_1} \rangle \rightarrow \langle S_b; e_{a_2} \rangle$ By E-Eval-Ctxt, it follows that $\langle S_b; E'_a[e_{a_1}] \rangle \rightarrow \langle S_b; E'_a[e_{a_2}] \rangle$, as we were required to show.

(c) Case E-Put: We have $S_b = S[l \mapsto d_1 \sqcup d_2]$. 

DRAFT: March 14, 2015

166
Choose $S' = S_b$, $i = 1, j = 1$, and $\pi = \text{id}$.

We have to show that:

- $\langle S; E'_b[e_{b_1}] \rangle \mapsto \langle S_b; E'_b[e_{b_2}] \rangle$, and
- $\langle S_b; E'_a[e_{a_1}] \rangle \mapsto \langle S_b; E'_a[e_{a_2}] \rangle$.

The first of these follows immediately from $\langle S; e_{b_1} \rangle \mapsto \langle S_b; e_{b_2} \rangle$ 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 $\langle S; e_{a_1} \rangle \mapsto \langle S_a; e_{a_2} \rangle$, we know that $[l \mapsto d_1 \sqcup d_2]$ is non-conflicting with it, and we know that $S \sqcup S[l \mapsto d_1 \sqcup d_2] \neq \top_S$ 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 $\langle S \sqcup S[l \mapsto d_1 \sqcup d_2]; e_{a_1} \rangle \mapsto \langle S_a \sqcup S[l \mapsto d_1 \sqcup d_2]; e_{a_2} \rangle$. Hence $\langle S_b; e_{a_1} \rangle \mapsto \langle S_b; e_{a_2} \rangle$. By E-Eval-Ctxt, it follows that $\langle S_b; E'_a[e_{a_1}] \rangle \mapsto \langle S_b; E'_a[e_{a_2}] \rangle$, as we were required to show.

(d) Case E-Put-Err:

Here $\langle S_b; e_{b_2} \rangle = \text{error}$, and so we choose $\sigma_c = \text{error}$, $i = 1, j = 0$, and $\pi = \text{id}$. We have to show that:

- $\langle S; E'_b[e_{b_1}] \rangle \mapsto \text{error}$, and
- $\langle S_b; E'_a[e_{a_1}] \rangle = \text{error}$.

The second of these is immediately true because since $\langle S_b; e_{b_2} \rangle = \text{error}$, $S_b = \top_S$, and so $\langle S_b; E'_a[e_{a_1}] \rangle$ is equal to $\text{error}$ as well. For the first, observe that $\langle S; e_{b_1} \rangle \mapsto \langle S_b; e_{b_2} \rangle$, hence by E-Eval-Ctxt, $\langle S; E'_b[e_{b_1}] \rangle \mapsto \langle S_b; E'_b[e_{b_2}] \rangle$. But $S_b = \top_S$, so $\langle S_b; E'_b[e_{b_2}] \rangle$ is equal to $\text{error}$, and so $\langle S; E'_b[e_{b_1}] \rangle \mapsto \text{error}$, 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 $S_b = S[l' \mapsto \bot]$. 

DRAFT: March 14, 2015
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 \( S_a = S[l \mapsto \bot] = S \cup_S [l \mapsto \bot] \), and that \( S[l' \mapsto \bot][l \mapsto \bot] = S[l' \mapsto \bot] \cup_S [l \mapsto \bot] \). Furthermore, since the only location allocated during the transition \( \langle S; e_{b_1} \rangle \longmapsto \langle S_b; e_{b_2} \rangle \) is \( l' \), we know that \( [l \mapsto \bot] \) is non-conflicting with it (since \( l \neq l' \) in this case). We also know that \( S[l' \mapsto \bot] \cup_S [l \mapsto \bot] \neq \top_S \), 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 \( \langle S \cup_S [l \mapsto \bot]; e_{b_1} \rangle \longmapsto \langle S_b \cup_S [l \mapsto \bot]; e_{b_2} \rangle \). Hence \( \langle S[l \mapsto \bot]; E'_b[e_{b_1}] \rangle \longmapsto \langle S_b[l \mapsto \bot]; E'_b[e_{b_2}] \rangle \). By E-Eval-Ctxt it follows that \( \langle S[l \mapsto \bot]; E'_a[e_{a_1}] \rangle \longmapsto \langle S_b[l \mapsto \bot]; E'_a[e_{a_2}] \rangle \), 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 \( l'' \notin \text{dom}(S) \) 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 \( \langle S[l'' \mapsto \bot][l \mapsto \bot]; E'_a[\pi(e_{a_2})] \rangle \).

  We have to show that:
A. PROOFS

- \( \langle S_a; E_b'[e_{b_1}] \rangle \longmapsto \langle S[l'' \mapsto \bot][l \mapsto \bot]; E_b'[\pi(e_{b_2})] \rangle \), 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 \( \langle S; e_{b_1} \rangle \longmapsto \langle S_b; e_{b_2} \rangle \), we have by Lemma 2.1 (Permutability) that \( \pi(\langle S; e_{b_1} \rangle) \longmapsto \pi(\langle S_b; e_{b_2} \rangle) \). 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 \( \langle S; e_{b_1} \rangle \longmapsto \langle S[l'' \mapsto \bot]; \pi(e_{b_2}) \rangle \).

Since the only location allocated during the transition \( \langle S; e_{b_1} \rangle \longmapsto \langle S[l'' \mapsto \bot]; \pi(e_{b_2}) \rangle \) is \( l'' \), we know that \( [l \mapsto \bot] \) is non-conflicting with it. We also know that \( S[l'' \mapsto \bot \downarrow S \downarrow l \mapsto \bot \] does not conflict with it. Therefore, by Lemma 2.5 (Independence), we have that \( \langle S \uplus S \downarrow S \downarrow l \mapsto \bot; e_{b_1} \rangle \longmapsto \langle S[l'' \mapsto \bot \downarrow S \downarrow l \mapsto \bot; \pi(e_{b_2}) \rangle \). Hence \( \langle S[l \mapsto \bot]; e_{b_1} \rangle \longmapsto \langle S[l'' \mapsto \bot \downarrow l \mapsto \bot; \pi(e_{b_2}) \rangle \). By E-Eval-Ctxt it follows that \( \langle S[l \mapsto \bot]; E_b'[e_{b_1}] \rangle \longmapsto \langle S[l'' \mapsto \bot \downarrow l \mapsto \bot]; E_b'[\pi(e_{b_2})] \rangle \), 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 \( \langle S[l'' \mapsto \bot]; (\pi(E_a'))[e_{a_1}] \rangle \longmapsto \langle S[l'' \mapsto \bot \downarrow l \mapsto \bot]; E_b'[\pi(e_{b_2})] \rangle \).

Since the only location allocated during the transition \( \langle S; e_{a_1} \rangle \longmapsto \langle S_a; e_{a_2} \rangle \) is \( l \), we know that \( [l'' \mapsto \bot] \) is non-conflicting with it. We also know that \( S_a \uplus S \downarrow S \downarrow l'' \mapsto \bot \) does not conflict with it. Therefore, by Lemma 2.5 (Independence), we have that \( \langle S \uplus S \downarrow S \downarrow l'' \mapsto \bot; e_{a_1} \rangle \longmapsto \langle S_a \uplus S \uplus S \downarrow l'' \mapsto \bot; e_{a_2} \rangle \). Hence \( \langle S[l'' \mapsto \bot]; e_{a_1} \rangle \longmapsto \langle S[l'' \mapsto \bot \downarrow l \mapsto \bot]; e_{a_2} \rangle \). By E-Eval-Ctxt it follows that \( \langle S[l'' \mapsto \bot]; (\pi(E_a'))[e_{a_1}] \rangle \longmapsto \langle S[l'' \mapsto \bot \downarrow l \mapsto \bot]; (\pi(E_a'))[e_{a_2}] \rangle \), which completes the case since \( E_b'[\pi(e_{b_2})] = (\pi(E_a'))[e_{a_2}] \).
(c) Case E-Put: We have $S_b = S[l' \mapsto d_1 \sqcup d_2]$, 

We have to show that:

- $\langle S_a; E'_b[e_{b_1}] \rangle \leftrightarrow \langle S_b[l \mapsto \bot]; E'_b[e_{b_2}] \rangle$, and
- $\langle S_b; E'_a[e_{a_1}] \rangle \leftrightarrow \langle S_b[l \mapsto \bot]; E'_a[e_{a_2}] \rangle$.

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 $\langle S; e_{b_1} \rangle \mapsto \langle S_b; e_{b_2} \rangle$, 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 $\langle S \sqcup_S [l \mapsto \bot]; e_{b_1} \rangle \mapsto \langle S_b \sqcup_S [l \mapsto \bot]; e_{b_2} \rangle$. Hence $\langle S[l \mapsto \bot]; e_{b_1} \rangle \mapsto \langle S_b[l \mapsto \bot]; e_{b_2} \rangle$.

By E-Eval-Ctxt, it follows that $\langle S[l \mapsto \bot]; E'_b[e_{b_1}] \rangle \mapsto \langle S_b[l \mapsto \bot]; E'_b[e_{b_2}] \rangle$, 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 $S_b[l \mapsto \bot] = S[l \mapsto \bot] \sqcup_S [l' \mapsto d_1 \sqcup d_2]$. Furthermore, since the only location allocated during the transition $\langle S; e_{a_1} \rangle \mapsto \langle S_a; e_{a_2} \rangle$ 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 $l' \in \dom(S)$, but we have from the side condition of E-New that $l \notin \dom(S)$.) 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 $\langle S \sqcup_S [l' \mapsto d_1 \sqcup d_2]; e_{a_1} \rangle \mapsto \langle S_a \sqcup_S [l' \mapsto d_1 \sqcup d_2]; e_{a_2} \rangle$. Hence $\langle S_b; e_{a_1} \rangle \mapsto \langle S_b[l \mapsto \bot]; e_{a_2} \rangle$. By E-Eval-Ctxt, it follows that $\langle S_b; E'_a[e_{a_1}] \rangle \mapsto \langle S_b[l \mapsto \bot]; E'_a[e_{a_2}] \rangle$, as we were required to show.

(d) Case E-Put-Err:

Here $\langle S_b; e_{b_2} \rangle = \text{error}$, and so we choose $\sigma_c = \text{error}$, $i = 1$, $j = 0$, and $\pi = \text{id}$. We have to show that:

- $\langle S_a; E'_b[e_{b_1}] \rangle \mapsto \text{error}$, and
A. PROOFS

- \( \langle S_b; E'_a[e_{a_1}] \rangle = \text{error} \).

The second of these is immediately true because \( \langle S_b; e_{b_2} \rangle = \text{error} \), \( S_b = \top_S \), and so \( \langle S_b; E'_a[e_{a_1}] \rangle \) is equal to error as well. For the first, observe that since \( \langle S; e_{a_1} \rangle \leftarrow \langle S; e_{a_2} \rangle \), we have by Lemma 2.4 (Monotonicity) that \( S \subseteq_S S_a \). Therefore, since \( \langle S; e_{b_1} \rangle \leftarrow \langle S; e_{b_1} \rangle \leftarrow \text{error} \), we have by Lemma 2.7 (Error Preservation) that \( \langle S_a; e_{b_1} \rangle \leftarrow \text{error} \). Since error is equal to \( \langle \top_S; e \rangle \) for all expressions \( e \), \( \langle S_a; e_{b_1} \rangle \leftarrow \langle \top_S; e \rangle \) for all \( e \). Therefore, by E-Eval-Ctxt, \( \langle S_a; E'_b[e_{b_1}] \rangle \leftarrow \langle \top_S; E'_b[e] \rangle \) for all \( e \). Since \( \langle \top_S; E'_b[e] \rangle \) is equal to error, we have that \( \langle S_a; E'_b[e_{a_1}] \rangle \leftarrow \text{error} \), 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 \( \pi = \text{id} \).

We have to show that:

- \( \langle S_a; E'_b[e_{b_1}] \rangle \leftarrow \langle S_a \sqcup S_b; E'_b[e_{b_2}] \rangle \), and

- \( \langle S_b; E'_a[e_{a_1}] \rangle \leftarrow \langle S_a \sqcup S_b; E'_b[e_{a_2}] \rangle \).

For the first of these, since no locations are allocated during the transition \( \langle S; e_{b_1} \rangle \leftarrow \langle S_b; e_{b_2} \rangle \), 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 \( \langle S \sqcup S [l \mapsto d_1 \sqcup d_2] ; e_{b_1} \rangle \leftarrow \langle S_b \sqcup S [l \mapsto d_1 \sqcup d_2] ; e_{b_2} \rangle \). By E-Eval-Ctxt, it follows that \( \langle S \sqcup S [l \mapsto d_1 \sqcup d_2] ; E'_b[e_{b_1}] \rangle \leftarrow \langle S_b \sqcup S [l \mapsto d_1 \sqcup d_2] ; E'_b[e_{b_2}] \rangle \). Since \( S \sqcup S [l \mapsto d_1 \sqcup d_2] = S[l \mapsto d_1 \sqcup d_2] = S_a \), we have that \( \langle S_a; E'_b[e_{b_1}] \rangle \leftarrow \langle S_b \sqcup_S \)
\[l \mapsto d_1 \sqcup d_2; \ E'_b[e_{b_1}]\]. Furthermore, since \(\langle S; \ e_{b_1} \rangle \longmapsto \langle S_b; \ e_{b_2} \rangle\), by Lemma 2.4 (Monotonicity), we have that \(S \subseteq S_b\), so \(S_b \sqcup S[l \mapsto d_1 \sqcup d_2] = S_b \sqcup S \sqcup S[l \mapsto d_1 \sqcup d_2] = S_b \sqcup S_a = S_a \sqcup S_b\). 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 store that is non-conflicting with \(\langle S; \ e_{a_1} \rangle \longmapsto \langle S_a; \ e_{a_2} \rangle\).

- \(S_b \sqcup S[l \mapsto d_1 \sqcup d_2] = \top_S\): Here we choose \(\sigma_c = \text{error}\) and \(\pi = \text{id}\). We have to show that there exist \(i \leq 1\) and \(j \leq 1\) such that:
  - \(\langle S_a; \ E'_b[e_{b_1}] \rangle \longmapsto^i \text{error}, \) and
  - \(\langle S_b; \ E'_b[e_{a_1}] \rangle \longmapsto^j \text{error}\).

For the first of these, since no locations are allocated during the transition \(\langle S; \ e_{b_1} \rangle \longmapsto \langle S_b; \ e_{b_2} \rangle\), 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 \(\langle S \sqcup S[l \mapsto d_1 \sqcup d_2]; \ e_{b_1} \rangle \longmapsto^{i'} \text{error}, \) where \(i' \leq 1\). Since \text{error} is equal to \(\langle \top_S; \ e \rangle\) for all expressions \(e\), \(\langle S \sqcup S[l \mapsto d_1 \sqcup d_2]; \ e_{b_1} \rangle \longmapsto^{i'} \langle \top_S; \ e \rangle\) 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 \text{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 \(\langle S_a; \ E'_b[e_{b_1}] \rangle \longmapsto \text{error}\), as required.

- If \(i' = 0\), we have that \(\langle S \sqcup S[l \mapsto d_1 \sqcup d_2]; \ e_{b_1} \rangle = \text{error}\). 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 \(\langle S_a; \ E'_b[e_{b_1}] \rangle = \text{error}\), as required.

The argument for the second is symmetrical, with \([l' \mapsto d'_1 \sqcup d'_2]\) being the store that is non-conflicting with \(\langle S; \ e_{a_1} \rangle \longmapsto \langle S_a; \ e_{a_2} \rangle\).
(d) Case E-Put-Err:

Here \( \langle S_b; e_{b_2} \rangle = \text{error} \), and so we choose \( \sigma_c = \text{error} \), \( i = 1 \), \( j = 0 \), and \( \pi = \text{id} \). We have to show that:

\[
\begin{aligned}
&\cdot \quad \langle S_a; E'_b[e_{a_1}] \rangle \nrightarrow \text{error}, \\
&\cdot \quad \langle S_b; E'_a[e_{a_1}] \rangle = \text{error}.
\end{aligned}
\]

The second of these is immediately true because since \( \langle S_b; e_{b_2} \rangle = \text{error} \), \( S_b = \top_S \), and so \( \langle S_b; E'_a[e_{a_1}] \rangle \) is equal to \( \text{error} \) as well. For the first, observe that since \( \langle S_a; e_{a_1} \rangle \nrightarrow \langle S_a; e_{a_2} \rangle \), we have by Lemma 2.4 (Monotonicity) that \( S_a \subseteq S_a \). Therefore, since \( \langle S; e_{b_1} \rangle \nrightarrow \text{error} \), we have by Lemma 2.7 (Error Preservation) that \( \langle S_a; e_{a_1} \rangle \nrightarrow \text{error} \). Since \( \text{error} \) is equal to \( \langle \top_S; e \rangle \) for all expressions \( e \), \( \langle S_a; e_{a_1} \rangle \nrightarrow \langle \top_S; e \rangle \) for all \( e \). Therefore, by E-Eval-Ctxt, \( \langle S_a; E'_b[e_{a_1}] \rangle \nrightarrow \langle \top_S; E'_b[e] \rangle \) for all \( e \). Since \( \langle \top_S; E'_b[e] \rangle \) is equal to \( \text{error} \), we have that \( \langle S_a; E'_b[e_{a_1}] \rangle \nrightarrow \text{error} \), 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 \( \langle S_a; e_{a_2} \rangle = \text{error} \).

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 \( \langle S_b; e_{b_2} \rangle = \text{error} \), and so we choose \( \sigma_c = \text{error} \), \( i = 0 \), \( j = 0 \), and \( \pi = \text{id} \). We have to show that:

\[
\begin{aligned}
&\cdot \quad \langle S_a; E'_b[e_{b_1}] \rangle = \text{error}, \\
&\cdot \quad \langle S_b; E'_a[e_{a_1}] \rangle = \text{error}.
\end{aligned}
\]

Since \( \langle S_a; e_{a_2} \rangle = \text{error} \), \( S_a = \top_S \), and since \( \langle S_b; e_{b_2} \rangle = \text{error} \), \( S_b = \top_S \), 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 \( S_{e_{b_1}} \) steps to \( S_{e_{b_2}} \):

(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 \( S_{b} = S \).

\( \square \)

A.8. Proof of Lemma 2.9

Proof. Suppose \( \sigma \rightarrow \sigma' \) and \( \sigma \rightarrow^{m} \sigma'' \), where \( 1 \leq m \). We have to show that there exist \( \sigma_c, i, j, \pi \) such that \( \sigma' \rightarrow^{i} \sigma_c \) and \( \pi(\sigma'') \rightarrow^{j} \sigma_c \) and \( i \leq m \) and \( j \leq 1 \).

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 \( \sigma \rightarrow^{m} \sigma'' \rightarrow^{m'} \sigma''' \) 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 \( \sigma_c, i, j, \pi \) such that \( \sigma' \rightarrow^{i} \sigma_c \) and \( \pi(\sigma'') \rightarrow^{j} \sigma_c \) and \( i \leq m + 1 \) and \( j \leq 1 \).

From the induction hypothesis, we have that there exist \( \sigma'_c, i', j', \pi' \) such that \( \sigma' \rightarrow^{i'} \sigma'_c \) and \( \pi'(\sigma'') \rightarrow^{j'} \sigma'_c \) and \( i' \leq m \) and \( j' \leq 1 \).

We proceed by cases on \( j' \):

- If \( j' = 0 \), then \( \pi'(\sigma'') = \sigma'_c \).

  Since \( \sigma'' \rightarrow \sigma''' \), we have that \( \pi'(\sigma'') \rightarrow \pi'(\sigma''') \) by Lemma 2.1 (Permutability).
A. PROOFS

We can then choose $\sigma_c = \pi'(\sigma''')$ and $i = i' + 1$ and $j = 0$ and $\pi = \pi'$. The key is that $\sigma' \mapsto i' \sigma_c' = \pi'(\sigma'') \mapsto \pi'(\sigma''')$ for a total of $i' + 1$ steps.

- If $j' = 1$:

  First, since $\pi'(\sigma'') \mapsto j' \sigma_c'$, then by Lemma 2.1 (Permutability) we have that $\sigma'' \mapsto i' \pi'^{-1}(\sigma_c')$.

  Then, by $\sigma'' \mapsto j' \pi'^{-1}(\sigma_c')$ and $\sigma'' \mapsto \sigma'''$ and Lemma 2.8 (Strong Local Confluence), we have that there exist $\sigma_c''$ and $i''$ and $j''$ such that $\pi'^{-1}(\sigma_c') \mapsto i'' \sigma_c''$ and $\pi''(\sigma''') \mapsto j'' \sigma_c''$ and $i'' \leq 1$ and $j'' \leq 1$.

  Since $\pi'^{-1}(\sigma_c') \mapsto i'' \sigma_c''$, by Lemma 2.1 (Permutability) we have that $\sigma_c'' \mapsto i'' \pi'(\sigma''')$.

  So we also have $\sigma' \mapsto j' \sigma_c' \mapsto i'' \pi'(\sigma''')$.

  Since $\pi''(\sigma''') \mapsto j'' \sigma_c''$, by Lemma 2.1 (Permutability) we have that $\pi'(\sigma''') \mapsto j'' \pi'(\sigma''')$.

  In summary, we pick $\sigma_c = \pi'(\sigma''')$ and $i = i' + i''$ and $j = j''$ and $\pi = \pi'' \circ \pi'$, which is sufficient because $i = i' + i'' \leq m + 1$ and $j = j'' \leq 1$.

\[\square\]

A.9. Proof of Lemma 2.10

Proof. Suppose that $\sigma \mapsto n \sigma'$ and $\sigma \mapsto m \sigma''$, where $1 \leq n$ and $1 \leq m$. We have to show that there exist $\sigma_c, i, j, \pi$ such that $\sigma' \mapsto i \sigma_c$ and $\pi(\sigma'') \mapsto j \sigma_c$ and $i \leq m$ and $j \leq n$.

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 $\sigma \mapsto n \sigma' \mapsto \sigma'''$ 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 $\sigma_c, i, j, \pi$ such that $\sigma''' \mapsto i \sigma_c$ and $\pi(\sigma''') \mapsto j \sigma_c$ and $i \leq m$ and $j \leq n + 1$.

From the induction hypothesis, we have that there exist $\sigma_c', i', j', \pi'$ such that $\sigma' \mapsto i' \sigma_c'$ and $\pi'(\sigma'') \mapsto j' \sigma_c'$ and $i' \leq m$ and $j' \leq n$. 

DRAFT: March 14, 2015 175
We proceed by cases on $i'$:

- If $i' = 0$, then $\sigma' = \sigma'_e$. We can then choose $\sigma''_e = \sigma''$ and $i = 0$ and $j = j' + 1$ and $\pi = \pi'$. Since $\pi'(\sigma'') \mapsto j'/\sigma'_e \mapsto \sigma''$, and $j' + 1 \leq n + 1$ since $j' \leq n$, the case is satisfied.

- If $i' \geq 1$:

From $\sigma' \mapsto \sigma''$ and $\sigma' \mapsto i'/\sigma'_e$ and Lemma 2.9, we have that there exist $\sigma''_e$ and $i''$ and $\pi''$ such that $\sigma''_e \mapsto i''/\sigma''$ and $\pi''(\sigma''_e) \mapsto j''/\sigma''$ and $i'' \leq i'$ and $j'' \leq 1$.

Since $\pi'(\sigma'') \mapsto j'/\sigma'_e$, by Lemma 2.1 (Permutability) we have that $\pi''(\pi'(\sigma'')) \mapsto j''/\sigma''$. We also have $\pi''(\pi'(\sigma'')) \mapsto j''/\sigma''$. So we also have $\pi''(\pi'(\sigma'')) \mapsto j''/\sigma''$. In summary, we pick $\sigma_e = \sigma''_e$ 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. Suppose that $(D, \sqsubseteq, \bot, \top)$ is a lattice and $(D_p, \sqsubseteq_p, \bot_p, \top_p) = \text{Freeze}(D, \sqsubseteq, \bot, \top)$. In order to show that $(D_p, \sqsubseteq_p, \bot_p, \top_p)$ is a lattice, we have to show that:

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

(2) Every nonempty finite subset of $D_p$ has a lub.

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

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

We prove each of these properties in turn:

(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.
A. PROOFS

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 $d_3 = \top$.
A. PROOFS

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 $= \subseteq$, $x_1 = x_3$.
  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 $\subseteq$, 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 lub.

To show this, it is sufficient to show that every two elements of $D_p$ have a lub, since a binary lub operation can be repeatedly applied to compute the lub of any finite set. We will show that every two elements of $D_p$ have a lub by showing that the $\sqcup_p$ operation defined by Definition 3.2 computes their lub.

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 $v_2 \sqsubseteq_p (v_1 \sqcup_p v_2)$.

Assume $v_1, v_2, v \in D_p$, if $v_1 \sqsubseteq_p v$ and $v_2 \sqsubseteq_p v$, then $v_1 \sqcup_p v_2 \sqsubseteq_p v$.

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 \( d_1 \sqcup d_2 \subseteq x \).

Hence \( (d_1 \sqcup d_2, \text{false}) \subseteq_p (x, \text{true}) \).

Hence \( v_1 \sqcup_p v_2 \subseteq_p v \).

- 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 \( (x_1, \text{true}) \subseteq_p (d, \text{false}) \), we know \( l = \top \).

  By inversion on \( (x_2, \text{true}) \subseteq_p (d, \text{false}) \), 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 \( \subseteq_p \), we have \((x_1, \text{true}) \subseteq_p (\top, \text{false})\). So \( v_1 \sqcup_p v_2 \subseteq_p v \).

  If it does not, then \( v_1 \sqcup_p v_2 = (\top, \text{false}) \).

  By the definition of \( \subseteq_p \), we have \((\top, \text{false}) \subseteq_p (\top, \text{false})\). So \( v_1 \sqcup_p v_2 \subseteq_p v \).

  -- Case \( v = (x, \text{true}) \):

  By inversion on \( (x_1, \text{true}) \subseteq_p (x, \text{true}) \), we know \( x = x_1 \).

  By inversion on \( (x_2, \text{true}) \subseteq_p (x, \text{true}) \), 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 \( v_1 \sqcup_p v_2 \subseteq_p v \).

- 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 \( d_2 \subseteq x_1 \).

  If it is, then \((x_1, \text{true}) \sqcup_p (d_2, \text{false}) = (x_1, \text{true}) = v_1 \).

  Hence \( v_1 \sqcup_p v_2 \subseteq_p v \).
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 definition of \(\sqsubseteq_p\), \((x_1, \text{true}) \sqsubseteq_p (x_1, \text{true})\).

  Hence \(v_1 \sqcup_p v_2 \sqsubseteq_p v\).

- Case \(v = (d_1, \text{false})\) and \(v = (x_2, \text{true})\):

  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 \(v_2 \sqsubseteq_p v_1 \sqcup v_2\).
A. PROOFS

• 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 \( v_1 \sqsubseteq_p v_1 \sqcup v_2 \).
    
    By reflexivity, \( (x_2, \text{true}) \sqsubseteq_p (x_2, \text{true}) \).
    
    Hence \( v_2 \sqsubseteq_p v_1 \sqcup v_2 \).
    
    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 \( (d_1, \text{false}) \sqsubseteq_p (\top, \text{false}) \).
    
    Hence \( v_1 \sqsubseteq_p v_1 \sqcup v_2 \).
    
    By the definition of \( \sqsubseteq_p \), we know \( (x_2, \text{true}) \sqsubseteq_p (\top, \text{false}) \).
    
    Hence \( v_2 \sqsubseteq_p v_1 \sqcup v_2 \).
    
    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 \( v_1 \sqsubseteq_p v_1 \sqcup v_2 \).
    
    By reflexivity, \( (x_2, \text{true}) \sqsubseteq_p (x_1, \text{true}) \).
Hence $v_2 \sqsubseteq_p v_1 \sqcup v_2$.

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 $(d, \text{false})$ for all $d \in D$, and $(x, \text{true})$ for all $x \in X$, where $X = D - \{\top\}$.

We consider both cases:

- $(d, \text{false})$ 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 $\bot_p = (\bot, \text{false}) \sqsubseteq_p (d, \text{false})$.

- $(x, \text{true})$ 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 $D_p$. 
A. PROOFS

⊤ is defined to be (⊤, 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 \( \sqsubseteq_p \), (\( d \), false) \( \sqsubseteq_p (\top, \text{false}) \) iff \( d \sqsubseteq \top \).
  - Since \( \top \) is the greatest element of \( D \), \( d \sqsubseteq \top \).
  - Therefore (\( d \), false) \( \sqsubseteq_p (\top, \text{false}) = \top_p \).

- (\( x \), true) for all \( x \in X \):
  - By the definition of \( \sqsubseteq_p \), (\( x \), true) \( \sqsubseteq_p (\top, \text{false}) \) iff \( \top \sqsubseteq \top \).
  - Therefore (\( x \), true) \( \sqsubseteq_p (\top, \text{false}) = \top_p \).

Therefore all elements of \( D_p \) are less than or equal to \( \top_p \).

\[ \square \]

A.11. Proof of Lemma 3.3

Proof. Consider an arbitrary permutation \( \pi \). For part 1, we have to show that if \( \sigma \rightsquigarrow \sigma' \) then \( \pi(\sigma) \rightsquigarrow \pi(\sigma') \), and that if \( \pi(\sigma) \rightsquigarrow \pi(\sigma') \) then \( \sigma \rightsquigarrow \sigma' \).

For the forward direction of part 1, suppose \( \sigma \rightsquigarrow \sigma' \). We have to show that \( \pi(\sigma) \rightsquigarrow \pi(\sigma') \). We proceed by cases on the rule by which \( \sigma \) steps to \( \sigma' \).

- Case E-Beta: \( \sigma = \langle S; (\lambda x. e) \, v \rangle \), and \( \sigma' = \langle S; e[x := v] \rangle \).
  - To show: \( \pi(\langle S; (\lambda x. e) \, v \rangle) \rightsquigarrow \pi(\langle S; e[x := v] \rangle) \).
  - By Definitions 3.11 and 3.9, \( \pi(\sigma) = \langle \pi(S); (\lambda x. \pi(e)) \pi(v) \rangle \).
  - 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 3.9, \( \langle \pi(S); \pi(e)[x := \pi(v)] \rangle \) 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 3.11. Hence the case is satisfied.

- Case E-New: \(\sigma = \langle S; \text{new} \rangle\), and \(\sigma' = \langle S[l \mapsto (\bot, \text{false})]; l \rangle\).

To show: \(\pi(\langle S; \text{new} \rangle) \longmapsto \pi(\langle S[l \mapsto (\bot, \text{false})]; l \rangle)\).

By Definitions 3.11 and 3.9, \(\pi(\sigma) = \langle \pi(S); \text{new} \rangle\).

By E-New, \(\langle \pi(S); \text{new} \rangle\) steps to \(\langle (\pi(S))[l' \mapsto (\bot, \text{false})]; l' \rangle\), where \(l' \not\in \text{dom}(\pi(S))\).

It remains to show that \(\langle (\pi(S))[l' \mapsto (\bot, \text{false})]; l' \rangle\) is equal to \(\pi(\langle S[l \mapsto (\bot, \text{false})]; l \rangle)\).

By Definition 3.11, \(\pi(\langle S[l \mapsto (\bot, \text{false})]; l \rangle)\) is equal to \(\pi(\langle S[l \mapsto (\bot, \text{false})]; \pi(l) \rangle)\), which is equal to \(\langle (\pi(S))[\pi(l) \mapsto (\bot, \text{false})]; \pi(l) \rangle\).

So, we have to show that \(\langle (\pi(S))[l' \mapsto (\bot, \text{false})]; l' \rangle\) is equal to \(\pi(\langle S[l \mapsto (\bot, \text{false})]; l \rangle)\).

Since we know (from the side condition of E-New) that \(l \not\in \text{dom}(S)\), it follows that \(\pi(l) \not\in \pi(\text{dom}(S))\).

Therefore, in \(\langle (\pi(S))[l' \mapsto (\bot, \text{false})]; l' \rangle\), we can \(\alpha\)-rename \(l'\) to \(\pi(l)\), and so the two configurations are equal and the case is satisfied.

- Case E-Put: \(\sigma = \langle S; \text{put}_i l \rangle\), and \(\sigma' = \langle S[l \mapsto u_{p_i}(p_1)]; \text{O} \rangle\).

To show: \(\pi(\langle S; \text{put}_i l \rangle) \longmapsto \pi(\langle S[l \mapsto u_{p_i}(p_1)]; \text{O} \rangle)\).

By Definition 3.11, \(\pi(\sigma) = \langle \pi(S); \text{put}_i \pi(l) \rangle\).

By E-Put, \(\langle \pi(S); \text{put}_i \pi(l) \rangle\) steps to \(\langle (\pi(S))[\pi(l) \mapsto u_{p_i}(p_1)]; \text{O} \rangle\), since \(S(l) = (\pi(S))(\pi(l)) = p_1\).

It remains to show that \(\langle (\pi(S))[\pi(l) \mapsto u_{p_i}(p_1)]; \text{O} \rangle\) is equal to \(\pi(\langle S[l \mapsto u_{p_i}(p_1)]; \text{O} \rangle)\).

By Definitions 3.11 and 3.9, \(\pi(\langle S[l \mapsto u_{p_i}(p_1)]; \text{O} \rangle)\) is equal to \(\langle (\pi(S))[\pi(l) \mapsto u_{p_i}(p_1)]; \text{O} \rangle\), and so the two configurations are equal and the case is satisfied.

- Case E-Put-Err: \(\sigma = \langle S; \text{put}_i l \rangle\), and \(\sigma' = \text{error}\).

To show: \(\pi(\langle S; \text{put}_i l \rangle) \longmapsto \pi(\text{error})\).

By Definition 3.11, \(\pi(\sigma) = \langle \pi(S); \text{put}_i \pi(l) \rangle\).

By E-Put-Err, \(\langle \pi(S); \text{put}_i \pi(l) \rangle\) steps to \(\text{error}\), since \(S(l) = (\pi(S))(\pi(l)) = p_1\).

Since \(\pi(\text{error}) = \text{error}\) by Definition 3.11, the case is complete.
A. PROOFS

- Case E-Get: \( \sigma = \langle S; \text{get} l \, P \rangle \), and \( \sigma' = \langle S; p_2 \rangle \).

  To show: \( \pi(\langle S; \text{get} l \, P \rangle) \iff \pi(\langle S; p_2 \rangle) \).

  By Definitions 3.11 and 3.9, \( \pi(\sigma) = \langle \pi(S); \text{get} \pi(l) \, P \rangle \).

  By E-Get, \( \langle \pi(S); \text{get} \pi(l) \, P \rangle \) steps to \( \langle \pi(S); p_2 \rangle \), since \( S(l) = \pi(S)(\pi(l)) = p_1 \).

  By Definitions 3.11 and 3.9, \( \pi(\langle S; p_2 \rangle) = \langle \pi(S); p_2 \rangle \). Therefore the case is complete.

- Case E-Freeze-Init: \( \sigma = \langle S; \text{freeze} l \, Q \, \lambda x. e \rangle \), and \( \sigma' = \langle S; \text{freeze} l \, Q \, \lambda x. e, \{\} , \{\} \rangle \).

  To show: \( \pi(\langle S; \text{freeze} l \, Q \, \lambda x. e \rangle) \iff \pi(\langle S; \text{freeze} l \, Q \, \lambda x. e, \{\} , \{\} \rangle) \).

  By Definitions 3.11 and 3.9, \( \pi(\sigma) = \langle \pi(S); \text{freeze} \pi(l) \, Q \, \lambda x. \pi(e) \rangle \).

  By E-Freeze-Init, \( \langle \pi(S); \text{freeze} \pi(l) \, Q \, \lambda x. \pi(e) \rangle \iff \langle \pi(S); \text{freeze} \pi(l) \, Q \, \lambda x. \pi(e) \rangle \).

  By Definitions 3.11 and 3.9, \( \pi(\langle S; \text{freeze} l \, Q \, \lambda x. e, \{\} , \{\} \rangle) = \pi(\langle S; \text{freeze} l \, Q \, \lambda x. e, \{\} , \{\} \rangle) \).

  Therefore the case is complete.

- Case E-Spawn-Handler: \( \sigma = \langle S; \text{freeze} l \, Q \, \lambda x. e_0, \{e, \ldots \} , H \rangle \), and \( \sigma' = \langle S; \text{freeze} l \, Q \, \lambda x. e_0, \{e_0[x := \text{true}] \} \cup H \rangle \).

  To show: \( \pi(\langle S; \text{freeze} l \, Q \, \lambda x. e_0, \{e, \ldots \} , H \rangle) \iff \pi(\langle S; \text{freeze} l \, Q \, \lambda x. e_0, \{e_0[x := \text{true}] \} \cup H \rangle) \).

  By Definitions 3.11 and 3.9, \( \pi(\sigma) = \langle \pi(S); \text{freeze} \pi(l) \, Q \, \lambda x. \pi(e_0), \{\pi(e_0)[x := \text{true}] \} \cup H \rangle \).

  Since \( \pi(S)(\pi(l)) = (d_1, \text{frz}_1) \), by E-Spawn-Handler we have that \( \langle \pi(S); \text{freeze} \pi(l) \, Q \, \lambda x. \pi(e_0), \{\pi(e_0)[x := \text{true}] \} \cup H \rangle \).

  By Definitions 3.11 and 3.9, \( \pi(\langle S; \text{freeze} l \, Q \, \lambda x. e_0, \{e_0[x := d_2], e, \ldots \} , \{d_2 \} \cup H \rangle) = \pi(\langle S; \text{freeze} l \, Q \, \lambda x. \pi(e_0), \{\pi(e_0)[x := \text{true}] \} \cup H \rangle) \).

  Therefore the case is complete.

- Case E-Freeze-Final: \( \sigma = \langle S; \text{freeze} l \, Q \, \lambda x. e_0, \{v, \ldots \} , H \rangle \), and \( \sigma' = \langle S[l \mapsto (d_1, \text{true})]; d_1 \rangle \).

  To show: \( \pi(\langle S; \text{freeze} l \, Q \, \lambda x. e_0, \{v, \ldots \} , H \rangle) \iff \pi(\langle S[l \mapsto (d_1, \text{true})]; d_1 \rangle) \).

  By Definitions 3.11 and 3.9, \( \pi(\sigma) = \langle \pi(S); \text{freeze} \pi(l) \, Q \, \lambda x. \pi(e_0), \{\pi(v), \ldots \} , H \rangle \).
LK: How do we handle applying \( \pi \) to an arbitrary value \( v \)? I’m just writing \( \pi(v) \) because we don’t know what kind of value it is.

Since \( (\pi(S))(\pi(l)) = (d_1, f r z_1) \), by E-Freeze-Final we have that \( \langle \pi(S); \text{freeze } \pi(l) \rangle \text{ after } Q \text{ with } \lambda x. \pi(e_0), \{ \pi(l) \} \).

We know from the forward direction of the proof that for all configurations \( S \) and \( l \) and \( \pi(l) \) and \( \pi(S) \), we have that \( \langle \pi(S); \text{freeze } \pi(l) \rangle \text{ after } Q \text{ with } \lambda x. \pi(e_0), \{ \pi(l) \} \), (From Definition 3.9, we can see that if \( v \) is a value, \( \pi(v) \) is also a value.)

It remains to show that \( \langle \pi(S)[\pi(l) \mapsto (d_1, true)]\rangle = \pi(\langle S[l \mapsto (d_1, true)]\rangle) \).

By Definitions 3.11 and 3.9, \( \pi(\langle S[l \mapsto (d_1, true)]\rangle) = \langle \pi(S)[\pi(l) \mapsto (d_1, true)]\rangle \).

We have to show that \( \langle \pi(S)[\pi(l) \mapsto (d_1, true)]\rangle \text{ is equal to } \pi(\langle S[l \mapsto (d_1, true)]\rangle) \).

By Definitions 3.11 and 3.9, \( \pi^T(S[l \mapsto (d_1, true)]) = \langle \pi(S)[\pi(l) \mapsto (d_1, true)]\rangle \).

For the reverse direction of part 1, suppose \( \pi(\sigma) \mapsto \pi(\sigma') \). We have to show that \( \sigma \mapsto \sigma' \).

We know from the forward direction of the proof that for all configurations \( \sigma \) and \( \sigma' \) and permutations \( \pi \), if \( \sigma \mapsto \sigma' \) then \( \pi(\sigma) \mapsto \pi(\sigma') \). Hence since \( \pi(\sigma) \mapsto \pi(\sigma') \), and since \( \pi^{-1} \) is also a permutation, we have that \( \pi^{-1}(\pi(\sigma)) \mapsto \pi^{-1}(\pi(\sigma')) \). Since \( \pi^{-1}(\pi(l)) = l \) for every \( l \in \text{Loc} \), and that property lifts to configurations as well, we have that \( \sigma \mapsto \sigma' \).

LK: Is the above enough of a proof?

For the forward direction of part 2, suppose \( \sigma \mapsto \sigma' \). We have to show that \( \pi(\sigma) \mapsto \pi(\sigma') \).
By inspection of the operational semantics, \( \sigma \) must be of the form \( \langle S; E[e] \rangle \), and \( \sigma' \) must be of the form \( \langle S'; E[e'] \rangle \). Hence we have to show that \( \pi(\langle S; E[e] \rangle) \mapsto \pi(\langle S'; E[e'] \rangle) \).

By Definition 3.11, \( \pi(\langle S; E[e] \rangle) \) 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 \( \langle \pi(S'); (\pi(E))[\pi(e')] \rangle \).

So we have to show that \( \langle \pi(S); (\pi(E))[\pi(e)] \rangle \mapsto \langle \pi(S'); (\pi(E))[\pi(e')] \rangle \).

From the premise of E-Eval-Ctxt, \( \langle S; e \rangle \mapsto \langle S'; e' \rangle \). Hence, by part 1, \( \pi(\langle S; e \rangle) \mapsto \pi(\langle S'; e' \rangle) \).

By Definition 3.11, \( \pi(\langle S; e \rangle) \) 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 \( \langle \pi(S); (\pi(e)) \rangle \mapsto \langle \pi(S'); (\pi(e')) \rangle \). Therefore, by E-Eval-Ctxt, \( \langle \pi(S); E[\pi(e)] \rangle \mapsto \langle \pi(S'); E[\pi(e')] \rangle \) for all evaluation contexts \( E \).

In particular, it is true that \( \langle \pi(S); (\pi(e))[\pi(e)] \rangle \mapsto \langle \pi(S'); (\pi(e))[\pi(e')] \rangle \), as we were required to show.

For the reverse direction of part 2, suppose \( \pi(\sigma) \mapsto \pi(\sigma') \). We have to show that \( \sigma \mapsto \sigma' \).

We know from the forward direction of the proof that for all configurations \( \sigma \) and \( \sigma' \) and permutations \( \pi \), if \( \sigma \mapsto \sigma' \) then \( \pi(\sigma) \mapsto \pi(\sigma') \). Hence since \( \pi(\sigma) \mapsto \pi(\sigma') \), and since \( \pi^{-1} \) is also a permutation, we have that \( \pi^{-1}(\pi(\sigma)) \mapsto \pi^{-1}(\pi(\sigma')) \). Since \( \pi^{-1}(\pi(l)) = l \) for every \( l \in Loc \), and that property lifts to configurations as well, we have that \( \sigma \mapsto \sigma' \).

**LK:** Is the above enough of a proof?

\[ \tag{\square} \]

### A.12. Proof of Lemma 3.4

Proof. Suppose \( \sigma \mapsto \sigma' \) and \( \sigma \mapsto \sigma'' \). We have to show that there is a permutation \( \pi \) such that \( \sigma' = \pi(\sigma'') \). The proof is by cases on the rule by which \( \sigma \) steps to \( \sigma' \).

- **Case E-Beta:**
Given: \( \langle S; (\lambda x. e) v \rangle \leadsto \langle S; e[x := v] \rangle \), and \( \langle S; (\lambda x. e) v \rangle \leadsto \sigma'' \).

To show: There exists a \( \pi \) 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 \( \pi \) to be the identity function.

- Case E-New:
  Given: \( \langle S; \text{new} \rangle \leadsto \langle S[l \mapsto (\bot, \text{false})]; l \rangle \), and \( \langle S; \text{new} \rangle \leadsto \sigma'' \).
  To show: There exists a \( \pi \) such that \( \langle S[l \mapsto (\bot, \text{false})]; l \rangle = \pi(\sigma'') \).

By inspection of the operational semantics, the only reduction rule by which \( \langle S; \text{new} \rangle \) can step is E-New. Hence \( \sigma'' = \langle S[l' \mapsto (\bot, \text{false})]; l' \rangle \). Since, by the side condition of E-New, neither \( l \) nor \( l' \) occur in \( \text{dom}(S) \), the case is satisfied by choosing \( \pi \) to be the permutation that maps \( l' \) to \( l \) and is the identity on every other element of \( \text{Loc} \).

- Case E-Put:
  Given: \( \langle S; \text{put}_{i_{l_1}} l \rangle \leadsto \langle S[l \mapsto u_{p_1}(p_1)]; 1 \rangle \), and \( \langle S; \text{put}_{i_{l_1}} l \rangle \leadsto \sigma'' \).
  To show: There exists a \( \pi \) such that \( \langle S[l \mapsto u_{p_1}(p_1)]; 1 \rangle = \pi(\sigma'') \).

By inspection of the operational semantics, and since \( u_{p_1}(p_1) = \top_p \) (from the premise of E-Put), the only reduction rule by which \( \langle S; \text{put}_{i_{l_1}} l \rangle \) can step is E-Put. Hence \( \sigma'' = \langle S[l \mapsto u_{p_1}(p_1)]; 1 \rangle \), and the case is satisfied by choosing \( \pi \) to be the identity function.

- Case E-Put-Err:
  Given: \( \langle S; \text{put}_{i_{l_1}} l \rangle \leadsto \text{error} \), and \( \langle S; \text{put}_{i_{l_1}} l \rangle \leadsto \sigma'' \).
  To show: There exists a \( \pi \) such that \( \text{error} = \pi(\sigma'') \). By inspection of the operational semantics, and since \( u_{p_1}(p_1) = \top_p \) (from the premise of E-Put-Err), the only reduction rule by which \( \langle S; \text{put}_{i_{l_1}} l \rangle \) can step is E-Put-Err. Hence \( \sigma'' = \text{error} \), and the case is satisfied by choosing \( \pi \) to be the identity function.

- Case E-Get:
  Given: \( \langle S; \text{get}_{l P} \rangle \leadsto \langle S; p_2 \rangle \), and \( \langle S; \text{get}_{l P} \rangle \leadsto \sigma'' \).
A. PROOFS

To show: There exists a $\pi$ such that $\langle S; p_2 \rangle = \pi(\sigma^n)$.

By inspection of the operational semantics, the only reduction rule by which $\langle S; \text{get } l \ P \rangle$ can step is E-Get. Hence $\sigma^n = \langle S; p_2 \rangle$, and the case is satisfied by choosing $\pi$ to be the identity function.

- Case E-Freeze-Init:

  Given: $\langle S; \text{freeze } l \ \text{after } Q \ \text{with } x: e \rangle \rightarrow \langle S; \text{freeze } l \ \text{after } Q \ \text{with } \lambda x. e. \{\}, \{\} \rangle$, and $\langle S; \text{freeze } l \ \text{after } Q \ \text{with } \lambda x. e. \rangle \rightarrow \sigma''$.

  To show: There exists a $\pi$ such that $\langle S; \text{freeze } l \ \text{after } Q \ \text{with } \lambda x. e. \{\}, \{\} \rangle = \pi(\sigma^n)$.

  By inspection of the operational semantics, the only reduction rule by which $\langle S; \text{freeze } l \ \text{after } Q \ \text{with } \lambda x. e. \rangle$ can step is E-Freeze-Init. Hence $\sigma'' = \langle S; \text{freeze } l \ \text{after } Q \ \text{with } \lambda x. e. \{\}, \{\} \rangle$, and the case is satisfied by choosing $\pi$ to be the identity function.

- Case E-Spawn-Handler:

  Given: $\langle S; \text{freeze } l \ \text{after } Q \ \text{with } x: e_0; f e; \ldots ; g; H \rangle \rightarrow \langle S; \text{freeze } l \ \text{after } Q \ \text{with } x: e_0; e_0[x := d_2]; e; \ldots ; g; f d_2; g; H \rangle$, and $\langle S; \text{freeze } l \ \text{after } Q \ \text{with } x: e_0; e_0[x := d_2]; e; \ldots ; g; H \rangle \rightarrow \sigma''$.

  To show: There exists a $\pi$ such that $\langle S; \text{freeze } l \ \text{after } Q \ \text{with } x: e_0; e_0[x := d_2]; e; \ldots ; g; f d_2; g; H \rangle = \pi(\sigma^n)$.

  By inspection of the operational semantics, the only reduction rule by which $\langle S; \text{freeze } l \ \text{after } Q \ \text{with } x: e_0; e, \ldots \rangle$ can step is E-Spawn-Handler. (It cannot step by E-Freeze-Final, because we have from the premises of E-Spawn-Handler that $d_2 \subseteq d_1$ and $d_2 \in Q$ and $d_2 \notin H$, and for the premises of E-Freeze-Final to hold, we would need that for all $d_2$, if $d_2 \subseteq d_1$ and $d_2 \in Q$, then $d_2 \in H$.) Hence $\sigma'' = \langle S; \text{freeze } l \ \text{after } Q \ \text{with } x: e_0; e_0[x := d_2]; e, \ldots ; g; f d_2; g; H \rangle$, and the case is satisfied by choosing $\pi$ to be the identity function.

- Case E-Freeze-Final:

  Given: $\langle S; \text{freeze } l \ \text{after } Q \ \text{with } x: e_0; v, \ldots \rangle, H \rightarrow \langle S[l \mapsto (d_1, \text{true})]; d_1 \rangle$, and $\langle S; \text{freeze } l \ \text{after } Q \ \text{with } x: e_0; v, \ldots \rangle, H \rightarrow \sigma''$.

  To show: There exists a $\pi$ such that $\langle S[l \mapsto (d_1, \text{true})]; d_1 \rangle = \pi(\sigma^n)$.
By inspection of the operational semantics, the only reduction rule by which \( \langle S; \text{freeze } l \text{ after } Q \text{ with } \lambda x. e_0, \{v_1, \ldots, v_n\} \rangle \) can step is E-Freeze-Final. (It cannot step by E-Spawn-Handler, because we have from the premises of E-Freeze-Final that, for all \( d_2 \subseteq d_1 \) and \( d_2 \in Q \), then \( d_2 \in H \), and for the premises of E-Spawn-Handler to hold, we would need that \( d_2 \subseteq d_1 \) and \( d_2 \in Q \) and \( d_2 \notin H \).) Hence \( \sigma'' = \langle S[l \mapsto (d_1, \text{true})]; d_1 \rangle \), and the case is satisfied by choosing \( \pi \) to be the identity function.

- **Case E-Freeze-Simple:**

  Given: \( \langle S; \text{freeze } l \rangle \xrightarrow{\text{E-Beta}} \langle S[l \mapsto (d_1, \text{true})]; d_1 \rangle \), and \( \langle S; \text{freeze } l \rangle \xrightarrow{\text{E-Spawn-Handler}} \sigma'' \).

  To show: There exists a \( \pi \) such that \( \langle S[l \mapsto (d_1, \text{true})]; d_1 \rangle = \pi(\sigma'') \).

  By inspection of the operational semantics, the only reduction rule by which \( \langle S; \text{freeze } l \rangle \) can step is E-Freeze-Simple. Hence \( \sigma'' = \langle S[l \mapsto (d_1, \text{true})]; d_1 \rangle \), and the case is satisfied by choosing \( \pi \) to be the identity function.

\[\square\]

A.13. **Proof of Lemma 3.5**

Proof. **TODO: Prove this.**

A.14. **Proof of Lemma 3.6**

Proof. Suppose \( \langle S; e \rangle \xrightarrow{\text{E-Beta}} \langle S'; e' \rangle \). We are required to show that \( S \trianglelefteq S' \). 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 \( \trianglelefteq \), since \( S \) does not change.

- **Case E-New:**

  Given: \( \langle S; \text{new} \rangle \xrightarrow{\text{E-New}} \langle S[l \mapsto (\bot, \text{false})]; l \rangle \).

  To show: \( S \trianglelefteq S[l \mapsto (\bot, \text{false})] \).
By Definition 3.6, we have to show that \( \text{dom}(S) \subseteq \text{dom}(S[l \mapsto (\bot, \text{false}]))) \) and that for all \( l' \in \text{dom}(S), S(l') \sqsubseteq_p (S[l \mapsto (\bot, \text{false}]))(l') \).

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 \( \text{dom}(S) \subseteq \text{dom}(S[l \mapsto (\bot, \text{false}]))) \).

From the side condition of E-New, \( l \not\in \text{dom}(S) \). Hence \( S[l \mapsto (\bot, \text{false}]]) \) adds a new binding for \( l \) in \( S \).

Hence \( S[l \mapsto (\bot, \text{false}]]) \) does not update any existing bindings in \( S \).

Hence, for all \( l' \in \text{dom}(S), S(l') \sqsubseteq_p (S[l \mapsto (\bot, \text{false}]))(l') \).

Therefore \( S \sqsubseteq S[l \mapsto (\bot, \text{false}]]) \), as required.

• Case E-Put:

Given: \( \langle S; \text{put}_i l \rangle \longmapsto \langle S[l \mapsto u_{p_i}(p_1)]; () \rangle \).

To show: \( S \sqsubseteq S[l \mapsto u_{p_i}(p_1)] \).

By Definition 3.6, we have to show that \( \text{dom}(S) \subseteq \text{dom}(S[l \mapsto u_{p_i}(p_1)]) \) and that for all \( l' \in \text{dom}(S), S(l') \sqsubseteq_p (S[l \mapsto u_{p_i}(p_1)])(l') \).

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 \( \text{dom}(S) \subseteq \text{dom}(S[l \mapsto u_{p_i}(p_1)]) \).

From the premises of E-Put, \( S(l) = p_1 \). Therefore \( l \in \text{dom}(S) \).

Hence \( S[l \mapsto u_{p_i}(p_1)] \) updates the existing binding for \( l \) in \( S \) from \( p_1 \) to \( u_{p_i}(p_1) \).

By definition, \( u_{p_i} \) is inflationary. Hence \( p_1 \sqsubseteq_p u_{p_i}(p_1) \).

\( S[l \mapsto u_{p_i}(p_1)] \) does not update any other bindings in \( S \), hence, for all \( l' \in \text{dom}(S), S(l') \sqsubseteq_p (S[l \mapsto u_{p_i}(p_1)])(l') \).

Hence \( S \sqsubseteq S[l \mapsto u_{p_i}(p_1)] \), as required.

• Case E-Put-Err:

Given: \( \langle S; \text{put}_i l \rangle \longmapsto \text{error} \).

By the definition of \text{error}, \text{error} = \langle \top_S; e \rangle \) for any \( e \).

To show: \( S \sqsubseteq \top_S \).
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: $⟨S; \text{freeze } l \text{ after } Q \text{ with } \lambda x. e_0, \{v, \ldots \}, H⟩ \longrightarrow ⟨S[l \mapsto (d_1, \text{true})]; d_1⟩$.  
  To show: $S \sqsubseteq_S S[l \mapsto (d_1, \text{true})]$.
  By Definition 3.6, we have to show that $\text{dom}(S) \subseteq \text{dom}(S[l \mapsto (d_1, \text{true})])$ and that for all $l' \in \text{dom}(S)$, $S(l') \sqsubseteq_p (S[l \mapsto (d_1, \text{true})])(l')$.
  
  LK: We could spell this out in even more excruciating detail, but I think it’s obvious enough.
  By definition, a store update operation on $S$ can only either update an existing binding in $S$ or extend $S$ with a new binding. Hence $\text{dom}(S) \subseteq \text{dom}(S[l \mapsto (d_1, \text{true})])$.
  From the premises of E-Freeze-Final, $S(l) = (d_1, \text{frz}_1)$. Therefore $l \in \text{dom}(S)$.
  Hence $S[l \mapsto (d_1, \text{true})]$ updates the existing binding for $l$ in $S$ from $(d_1, \text{frz}_1)$ to $(d_1, \text{true})$.
  By the definition of $\sqsubseteq_p$, $(d_1, \text{frz}_1) \sqsubseteq_p (d_1, \text{true})$.
  $S[l \mapsto (d_1, \text{true})]$ does not update any other bindings in $S$, hence, for all $l' \in \text{dom}(S)$, $S(l') \sqsubseteq_p (S[l \mapsto (d_1, \text{true})])(l')$.
  Hence $S \sqsubseteq_S S[l \mapsto (d_1, \text{true})]$, as required.

• Case E-Freeze-Simple:
  Given: $⟨S; \text{freeze } l⟩ \longrightarrow ⟨S[l \mapsto (d_1, \text{true})]; d_1⟩$.
  To show: $S \sqsubseteq_S S[l \mapsto (d_1, \text{true})]$.
  Similar to the previous case.
A.15. Proof of Lemma 3.7

Proof. Suppose \( \langle S; e \rangle \longrightarrow \langle S'; e' \rangle \), where \( \langle S'; e' \rangle \neq \text{error} \). Consider arbitrary \( U_S \) such that \( U_S \) is non-conflicting with \( \langle S; e \rangle \longrightarrow \langle S'; e' \rangle \) and \( U_S(S') \neq \top_S \) and \( U_S \) is freeze-safe with \( \langle S; e \rangle \longrightarrow \langle S'; e' \rangle \). We are required to show that \( \langle U_S(S); e \rangle \longrightarrow \langle U_S(S); e' \rangle \).

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 \( \langle S'; e' \rangle \neq \text{error} \), we do not need to consider the E-Put-Err rule. The assumption that \( U_S \) is freeze-safe with \( \langle S; e \rangle \longrightarrow \langle S'; e' \rangle \) is only needed in the E-Freeze-Final and E-Freeze-Simple cases.

- **Case E-Beta:**
  
  Given: \( \langle S; (\lambda x. e) v \rangle \longrightarrow \langle S; e[x := v] \rangle \).
  
  To show: \( \langle U_S(S); (\lambda x. e) v \rangle \longrightarrow \langle U_S(S); e[x := v] \rangle \).
  
  Immediate by E-Beta.

- **Case E-New:**
  
  Given: \( \langle S; \text{new} \rangle \longrightarrow \langle S[l \mapsto (\bot, \text{false})]; l \rangle \).
  
  To show: \( \langle U_S(S); \text{new} \rangle \longrightarrow \langle U_S(S[l \mapsto (\bot, \text{false})]); l \rangle \).
  
  By E-New, we have that \( \langle U_S(S); \text{new} \rangle \longrightarrow \langle (U_S(S))[l' \mapsto (\bot, \text{false})]; l' \rangle \), where \( l' \notin \text{dom}(U_S(S)) \).
  
  By assumption, \( U_S \) is non-conflicting with \( \langle S; \text{new} \rangle \longrightarrow \langle S[l \mapsto (\bot, \text{false})]; l \rangle \). Therefore \( l \notin \text{dom}(U_S(S)) \).
  
  Therefore, in \( \langle (U_S(S))[l' \mapsto (\bot, \text{false})]; l' \rangle \), we can \( \alpha \)-rename \( l' \) to \( l \).
  
  Therefore \( \langle U_S(S); \text{new} \rangle \longrightarrow \langle (U_S(S))[l \mapsto (\bot, \text{false})]; l \rangle \).
  
  Also, since \( U_S \) is non-conflicting with \( \langle S; \text{new} \rangle \longrightarrow \langle S[l \mapsto (\bot, \text{false})]; l \rangle \), we have that \( (U_S(S[l \mapsto (\bot, \text{false})]))(l) = (U_S(S[l \mapsto (\bot, \text{false})]))(l) = (\bot, \text{false}) \).
  
  Hence \( (U_S(S))[l \mapsto (\bot, \text{false})] = U_S(S[l \mapsto (\bot, \text{false})]) \).
  
  Therefore \( \langle U_S(S); \text{new} \rangle \longrightarrow \langle U_S(S[l \mapsto (\bot, \text{false})]); l \rangle \), as we were required to show.

- **Case E-Put:**
  
  Given: \( \langle S; \text{put}_i l \rangle \longrightarrow \langle S[l \mapsto u_{p_i}(p_1)]; () \rangle \).
To show: \( \langle U_S(S); \text{put}_i l \rangle \leftrightarrow \langle U_S(S[l \mapsto u_{p_i}(p_1)]); \top \rangle \).

From the premises of E-Put, \( S(l) = p_1 \).

Hence \( (U_S(S))(l) = p'_1 \), where \( p_1 \sqsubseteq_p p'_1 \).

Next, we want to show that \( u_{p_i}(p'_1) \neq \top_p \).

Assume for the sake of a contradiction that \( u_{p_i}(p'_1) = \top_p \).

Then \( u_{p_i}((U_S(S))(l)) = \top_p \).

Let \( u_{p_j} \) be the state update operation in \( U_S \) that affects the contents of \( l \). Hence \( (U_S(S))(l) = u_{p_j}(p_1) \). Then \( u_{p_i}(u_{p_j}(p_1)) = \top_p \).

Since state update operations commute, \( u_{p_j}(u_{p_i}(p_1)) = \top_p \).

But then \( U_S(S[l \mapsto u_{p_i}(p_1)]) = \top_S \), which contradicts the assumption that \( U_S(S[l \mapsto u_{p_i}(p_1)]) \neq \top_S \).

Hence, \( u_{p_i}(p'_1) \neq \top_p \).

Therefore, by E-Put, \( \langle U_S(S); \text{put}_i l \rangle \leftrightarrow \langle (U_S(S))[l \mapsto u_{p_i}(p'_1)]; \top \rangle \).

Since \( p'_1 = u_{p_j}(p_1) \), we have that \( (U_S(S))[l \mapsto u_{p_i}(p'_1)] = (U_S(S))[l \mapsto u_{p_i}(u_{p_j}(p_1))] \), which, since \( u_{p_i} \) and \( u_{p_j} \) commute, is equal to \( (U_S(S))[l \mapsto u_{p_j}(u_{p_i}(p_1))] \).

Finally, since \( u_{p_j} \) is the update operation in \( U_S \) that affects the contents of \( l \), we have that \( (U_S(S))[l \mapsto u_{p_j}(u_{p_i}(p_1))] = U_S(S[l \mapsto u_{p_i}(p_1)]) \), and so the case is satisfied.

- Case E-Get:

Given: \( \langle S; \text{get} P \rangle \leftrightarrow \langle S; p_2 \rangle \).

To show: \( \langle U_S(S); \text{get} P \rangle \leftrightarrow \langle U_S(S); p_2 \rangle \).

From the premises of E-Get, \( S(l) = p_1 \) and \( \text{incomp}(P) \) and \( p_2 \in P \) and \( p_2 \sqsubseteq_p p_1 \).

By assumption, \( U_S(S) \neq \top_S \).

Hence \( (U_S(S))(l) = p'_1 \), where \( p_1 \sqsubseteq_p p'_1 \).

By the transitivity of \( \sqsubseteq_p \), \( p_2 \sqsubseteq_p p'_1 \).

Hence, \( (U_S(S))(l) = p'_1 \) and \( \text{incomp}(P) \) and \( p_2 \in P \) and \( p_2 \sqsubseteq_p p'_1 \).

Therefore, by E-Get,
\[
\langle U_S(S); \text{get } \ell \, P \rangle \rightarrow \langle U_S(S); \, p_2 \rangle,
\]
as we were required to show.

- **Case E-Freeze-Init:**
  
  Given: \(\langle S; \text{freeze } \ell \, \text{after } Q \, \text{with } \lambda x. \, e \rangle \rightarrow \langle S; \text{freeze } \ell \, \text{after } Q \, \text{with } \lambda x. \, e, \{\}, \{\} \rangle\).
  
  To show: \(\langle U_S(S); \text{freeze } \ell \, \text{after } Q \, \text{with } \lambda x. \, e \rangle \rightarrow \langle U_S(S); \text{freeze } \ell \, \text{after } Q \, \text{with } \lambda x. \, e, \{\}, \{\} \rangle\).
  
  Immediate by E-Freeze-Init.

- **Case E-Spawn-Handler:**
  
  Given:
  
  \[
  \langle S; \text{freeze } \ell \, \text{after } Q \, \text{with } \lambda x. \, e_0, \{e, \ldots \}, H \rangle \rightarrow \langle S; \text{freeze } \ell \, \text{after } Q \, \text{with } \lambda x. \, e_0, \{e_0[x := d_2], e, \ldots \}, \{\} \rangle.
  \]
  
  To show:
  
  \[
  \langle U_S(S); \text{freeze } \ell \, \text{after } Q \, \text{with } \lambda x. \, e_0, \{e, \ldots \}, H \rangle \rightarrow \langle U_S(S); \text{freeze } \ell \, \text{after } Q \, \text{with } \lambda x. \, e_0, \{e_0[x := d_2], e, \ldots \}, \{\} \rangle.
  \]
  
  From the premises of E-Spawn-Handler, \(S(\ell) = (d_1, \text{frz}_1)\) and \(d_2 \subseteq d_1\) and \(d_2 \not\subseteq H\) and \(d_2 \in Q\).
  
  By assumption, \(U_S(S) \neq \top_S\).
  
  Hence \((U_S(S))(\ell) = (d'_1, \text{frz}'_1)\) where \((d_1, \text{frz}_1) \sqsubseteq_p (d'_1, \text{frz}'_1)\).
  
  By Definition 3.1, \(d_1 \subseteq d'_1\).
  
  By the transitivity of \(\subseteq\), \(d_2 \subseteq d'_1\).
  
  Hence \((U_S(S))(\ell) = (d'_1, \text{frz}'_1)\) and \(d_2 \subseteq d'_1\) and \(d_2 \not\subseteq H\) and \(d_2 \in Q\).
  
  Therefore, by E-Spawn-Handler,
  
  \[
  \langle U_S(S); \text{freeze } \ell \, \text{after } Q \, \text{with } \lambda x. \, e_0, \{e, \ldots \}, H \rangle \rightarrow \langle U_S(S); \text{freeze } \ell \, \text{after } Q \, \text{with } \lambda x. \, e_0, \{e_0[x := d_2], e, \ldots \}, \{\} \rangle.
  \]
  
  as we were required to show.

- **Case E-Freeze-Final:**
  
  Given: \(\langle S; \text{freeze } \ell \, \text{after } Q \, \text{with } \lambda x. \, e_0, \{v, \ldots \}, H \rangle \rightarrow \langle S[\ell \mapsto (d_1, \text{true})]; d_1 \rangle\).
  
  To show: \(\langle U_S(S); \text{freeze } \ell \, \text{after } Q \, \text{with } \lambda x. \, e_0, \{v, \ldots \}, H \rangle \rightarrow \langle U_S(S[\ell \mapsto (d_1, \text{true})]); d_1 \rangle\).
From the premises of E-Freeze-Final, \( S(l) = (d_1, \text{frz}_1) \).

We have two cases to consider:

- \( \text{frz}_1 = \text{true} \):
  In this case, \( S(l) = (d_1, \text{true}) \).

  Let \( u_{p_i} \) be the state update operation in \( U_S \) that affects the contents of \( l \). Hence \((U_S(S))(l) = u_{p_i}((d_1, \text{true})) \).

  We know from Definition 3.4 that \( u_{p_i}((d_1, \text{true})) \) is either \((d_1, \text{true})\) or \((\top, \text{false})\).

  But if \( u_{p_i}((d_1, \text{true})) = (\top, \text{false}) \), then \( U_S(S[l \mapsto (d_1, \text{true})]) = \top \), which contradicts our assumption that \( U_S(S[l \mapsto (d_1, \text{true})]) \neq \top \). Hence \( u_{p_i}((d_1, \text{true})) = (d_1, \text{true}) \).

  Hence \((U_S(S))(l) = (d_1, \text{true})\), and we already have from the premises of E-Freeze-Final that \( \forall d_2 . (d_2 \subseteq d_1 \land d_2 \in Q \Rightarrow d_2 \in H) \). Hence, by E-Freeze-Final, we have that \( \langle U_S(S); \text{freeze } l \text{ after } Q \text{ with } \lambda x. e_0, \lbrace v, \ldots \rbrace, H \rangle \longrightarrow \langle S[l \mapsto (d_1, \text{true})] \rangle; d_1 \rangle \).

  Finally, since \( u_{p_i} \) is the state update operation in \( U_S \) that affects the contents of \( l \), and \( u_{p_i}((d_1, \text{true})) = (d_1, \text{true}) \), we have that \( (U_S(S))[l \mapsto (d_1, \text{true})] \) is equal to \( U_S(S[l \mapsto (d_1, \text{true})]) \), and so the case is satisfied.

- \( \text{frz}_1 = \text{false} \):
  By assumption, \( U_S \) is freeze-safe with \( \langle S; \text{freeze } l \text{ after } Q \text{ with } \lambda x. e_0, \lbrace v, \ldots \rbrace, H \rangle \longrightarrow \langle S[l \mapsto (d_1, \text{true})] \rangle; d_1 \rangle \). Therefore \( U_S \) acts as the identity on the contents of any locations that change status during the transition. Since \( \text{frz}_1 = \text{false} \), the contents of \( l \) change status during the transition. Therefore \( U_S \) acts as the identity on the contents of \( l \).

  Hence \((U_S(S))(l) = S(l) = (d_1, \text{frz}_1)\), and we already have from the premises of E-Freeze-Final that \( \forall d_2 . (d_2 \subseteq d_1 \land d_2 \in Q \Rightarrow d_2 \in H) \). Hence, by E-Freeze-Final, we have that \( \langle U_S(S); \text{freeze } l \text{ after } Q \text{ with } \lambda x. e_0, \lbrace v, \ldots \rbrace, H \rangle \longrightarrow \langle (U_S(S))[l \mapsto (d_1, \text{true})] \rangle; d_1 \rangle \).

  Finally, since \( U_S \) acts as the identity on the contents of \( l \), we have that \( (U_S(S))[l \mapsto (d_1, \text{true})] \) is equal to \( U_S(S[l \mapsto (d_1, \text{true})]) \), and so the case is satisfied.

- Case E-Freeze-Simple:
A. PROOFS

Given: \( <S; \text{freeze } l> \leftarrow <S[l \mapsto (d_1, \text{true})]; d_1> \).

To show: \( <U_S(S); \text{freeze } l> \leftarrow <U_S(S[l \mapsto (d_1, \text{true})]); d_1> \).

From the premises of E-Freeze-Simple, \( S(l) = (d_1, \text{frz}_1) \).

We have two cases to consider:

- \( \text{frz}_1 = \text{true} \):
  
  In this case, \( S(l) = (d_1, \text{true}) \).

  Let \( u_{pi} \) be the state update operation in \( U_S \) that affects the contents of \( l \). Hence \( (U_S(S))(l) = u_{pi}((d_1, \text{true})) \).

  We know from Definition 3.4 that \( u_{pi}((d_1, \text{true})) \) is either \( (d_1, \text{true}) \) or \( (\top, \text{false}) \).

  But if \( u_{pi}((d_1, \text{true})) = (\top, \text{false}) \), then \( U_S(S[l \mapsto (d_1, \text{true})]) = \top_S \), which contradicts our assumption that \( U_S(S[l \mapsto (d_1, \text{true})]) \neq \top_S \). Hence \( u_{pi}((d_1, \text{true})) = (d_1, \text{true}) \).

  Hence \( (U_S(S))(l) = (d_1, \text{true}) \). Hence, by E-Freeze-Simple, we have that \( <U_S(S); \text{freeze } l> \leftarrow <(U_S(S))(l) \mapsto (d_1, \text{true})]; d_1> \).

  Finally, since \( U_S \) acts as the identity on the contents of \( l \), we have that \( (U_S(S))(l \mapsto (d_1, \text{true})) \) is equal to \( U_S(S[l \mapsto (d_1, \text{true})]) \), and so the case is satisfied.

- \( \text{frz}_1 = \text{false} \):
  
  By assumption, \( U_S \) is freeze-safe with \( <S; \text{freeze } l> \leftarrow <S[l \mapsto (d_1, \text{true})]; d_1> \). Therefore \( U_S \) acts as the identity on the contents of any locations that change status during the transition. Since \( \text{frz}_1 = \text{false} \), the contents of \( l \) change status during the transition. Therefore \( U_S \) acts as the identity on the contents of \( l \).

  Hence \( (U_S(S))(l) = S(l) = (d_1, \text{frz}_1) \). Hence, by E-Freeze-Simple, we have that \( <U_S(S); \text{freeze } l> \leftarrow <(U_S(S))(l) \mapsto (d_1, \text{true})]; d_1> \).

  Finally, since \( U_S \) acts as the identity on the contents of \( l \), we have that \( (U_S(S))(l \mapsto (d_1, \text{true})) \) is equal to \( U_S(S[l \mapsto (d_1, \text{true})]) \), and so the case is satisfied.

\[ \square \]
A.16. Proof of Lemma 3.8

Proof. Suppose \( \langle S; e \rangle \leadsto \langle S'; e' \rangle \), where \( \langle S'; e' \rangle \neq \text{error} \). Consider arbitrary \( U_S \) such that \( U_S \) is non-conflicting with \( \langle S; e \rangle \leadsto \langle S'; e' \rangle \) and \( U_S(S') = \top_S \) and \( U_S \) is freeze-safe with \( \langle S; e \rangle \leadsto \langle S'; e' \rangle \). We are required to show that there exists \( i \leq 1 \) such that \( \langle U_S(S); e \rangle \leadsto^i \text{error} \).

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 \( \langle S'; e' \rangle \neq \text{error} \), we do not need to consider the E-Put-Err rule. The assumption that \( U_S \) is freeze-safe with \( \langle S; e \rangle \leadsto \langle S'; e' \rangle \) is only needed in the E-Freeze-Final and E-Freeze-Simple cases.

- Case E-Beta:
  
  Given: \( \langle S; (\lambda x. e) v \rangle \leadsto \langle S; e[x := v] \rangle \).

  To show: \( \langle U_S(S); (\lambda x. e) v \rangle \leadsto \text{error} \), where \( i \leq 1 \).

  By assumption, \( U_S(S') = \top_S \). Since \( S' = S \), it must be the case that \( U_S(S') = U_S(S) = \top_S \).

  Hence, by the definition of \( \text{error} \), \( \langle U_S(S); (\lambda x. e) v \rangle = \text{error} \).

  Hence \( \langle U_S(S); (\lambda x. e) v \rangle \leadsto^i \text{error} \), with \( i = 0 \).

- Case E-New:
  
  Given: \( \langle S; \text{new} \rangle \leadsto \langle S[l \mapsto (\bot, \text{false})]; l \rangle \).

  To show: \( \langle U_S(S); \text{new} \rangle \leadsto^i \text{error} \), where \( i \leq 1 \).

  By E-New, \( \langle U_S(S); \text{new} \rangle \leadsto \langle (U_S(S))[l' \mapsto (\bot, \text{false})]; l' \rangle \), where \( l' \notin \text{dom}(U_S(S)) \).

  By assumption, \( U_S \) is non-conflicting with \( \langle S; \text{new} \rangle \leadsto \langle S[l \mapsto (\bot, \text{false})]; l \rangle \). Therefore \( l \notin \text{dom}(U_S(S)) \).

  Therefore, in \( \langle (U_S(S))[l' \mapsto (\bot, \text{false})]; l' \rangle \), we can \( \alpha \)-rename \( l' \) to \( l \).

  Therefore \( \langle U_S(S); \text{new} \rangle \leadsto \langle (U_S(S))[l \mapsto (\bot, \text{false})]; l \rangle \).

  Also, since \( U_S \) is non-conflicting with \( \langle S; \text{new} \rangle \leadsto \langle S[l \mapsto (\bot, \text{false})]; l \rangle \), we have that \( U_S(S[l \mapsto (\bot, \text{false})])[l \mapsto (\bot, \text{false})]) = (S[l \mapsto (\bot, \text{false})])[l \mapsto (\bot, \text{false})] \).

  Hence \( U_S(S)[l \mapsto (\bot, \text{false})] = U_S(S[l \mapsto (\bot, \text{false})]) \).

  Therefore \( \langle U_S(S); \text{new} \rangle \leadsto \langle U_S(S[l \mapsto (\bot, \text{false})]); l \rangle \).
By assumption, \( U_S(S[l \mapsto (\bot, \text{false})]) = \top_S \).

Therefore \( \langle U_S(S); \text{new} \rangle \longmapsto \langle \top_S; l \rangle \).

Hence, by the definition of error, \( \langle U_S(S); \text{new} \rangle \longmapsto \text{error} \).

Hence \( \langle U_S(S); \text{new} \rangle \longmapsto \text{error} \), with \( i = 1 \).

Case E-Put:

Given: \( \langle S; \text{put}_i l \rangle \longmapsto \langle S[l \mapsto u_p(p_1)]; \circ \rangle \).

To show: \( \langle U_S(S); \text{put}_i l \rangle \longmapsto \text{error} \), where \( i' \leq 1 \).

Consider whether \( U_S(S) = \top_S \):

- If \( U_S(S) = \top_S \):

  In this case, by the definition of error, \( \langle U_S(S); \text{put}_i l \rangle = \text{error} \).

  Hence \( \langle U_S(S); \text{put}_i l \rangle \longmapsto \text{error} \), with \( i' = 0 \).

- If \( U_S(S) \neq \top_S \):

  Since \( U_S(S) \neq \top_S \), we know that \( S \neq \top_S \). Also, from the premises of E-Put, we have that \( u_p(p_1) \neq \top_p \). Hence \( S[l \mapsto u_p(p_1)] \neq \top_S \).

  Since \( U_S(S) \neq \top_S \) and \( S[l \mapsto u_p(p_1)] \neq \top_S \), but \( U_S(S[l \mapsto u_p(p_1)]) = \top_S \), it must be \( U_S \)'s action on the contents of \( l \) that updates \( S[l \mapsto u_p(p_1)] \) to \( \top_S \).

  Let \( u_{p_j} \) be the state update operation in \( U_S \) that affects the contents of \( l \).

  Then \( u_{p_j}(u_p(p_1)) = \top_p \).

  Since state update operations commute, \( u_p(u_{p_j}(p_1)) = \top_p \).

  Since \( u_{p_j} \) is the state update operation in \( U_S \) that affects the contents of \( l \), we have that \( (U_S(S))(l) = u_{p_j}(p_1) \).

  Since \( U_S(S) \neq \top_S \), \( u_{p_j}(p_1) \neq \top_p \).

  Therefore, by E-Put, \( \langle U_S(S); \text{put}_i l \rangle \longmapsto \langle (U_S(S))[l \mapsto u_p(u_{p_j}(p_1))]; \circ \rangle \).

  Since \( u_{p_j}(u_p(p_1)) = \top_p \), \( \langle U_S(S); \text{put}_i l \rangle \longmapsto \text{error} \).

  Hence \( \langle U_S(S); \text{put}_i l \rangle \longmapsto \text{error} \), with \( i' = 1 \).

Case E-Get:
Given: \( \langle S; \text{get } l P \rangle \rightarrow \langle S; p_2 \rangle \).

To show: \( \langle U_S(S); \text{get } l P \rangle \rightarrow^i \text{error} \), where \( i \leq 1 \).

By assumption, \( U_S(S) = \top \).

Hence, by the definition of \text{error}, \( \langle U_S(S); \text{get } l P \rangle = \text{error} \).

Hence \( \langle U_S(S); \text{get } l P \rangle \rightarrow^i \text{error} \), with \( i = 0 \).

- Case E-Freeze-Init:
  
  Given: \( \langle S; \text{freeze } l \text{ after } Q \text{ with } \lambda x. e \rangle \rightarrow \langle S; \text{freeze } l \text{ after } Q \text{ with } \lambda x. e, \{\}, \{\} \rangle \).

  To show: \( \langle U_S(S); \text{freeze } l \text{ after } Q \text{ with } \lambda x. e \rangle \rightarrow^i \text{error} \), where \( i \leq 1 \).

  By assumption, \( U_S(S) = \top \).

  Hence, by the definition of \text{error}, \( \langle U_S(S); \text{freeze } l \text{ after } Q \text{ with } \lambda x. e \rangle = \text{error} \).

  Hence \( \langle U_S(S); \text{freeze } l \text{ after } Q \text{ with } \lambda x. e \rangle \rightarrow^i \text{error} \), with \( i = 0 \).

- Case E-Spawn-Handler:
  
  Given:

  \[
  \langle S; \text{freeze } l \text{ after } Q \text{ with } \lambda x. e_0, \{e, \ldots \}, H \rangle \rightarrow \langle S; \text{freeze } l \text{ after } Q \text{ with } \lambda x. e_0, \{e_0[x := d_2], e, \ldots \}, \{\}, \{\} \rangle.
  \]

  To show:

  \( \langle U_S(S); \text{freeze } l \text{ after } Q \text{ with } \lambda x. e_0, \{e, \ldots \}, H \rangle \rightarrow^i \text{error} \), where \( i \leq 1 \).

  By assumption, \( U_S(S) = \top \).

  Hence, by the definition of \text{error}, \( \langle U_S(S); \text{freeze } l \text{ after } Q \text{ with } \lambda x. e_0, \{e, \ldots \}, H \rangle = \text{error} \).

  Hence \( \langle U_S(S); \text{freeze } l \text{ after } Q \text{ with } \lambda x. e_0, \{e, \ldots \}, H \rangle \rightarrow^i \text{error} \), with \( i = 0 \).

- Case E-Freeze-Final:
  
  Given: \( \langle S; \text{freeze } l \text{ after } Q \text{ with } \lambda x. e_0, \{v, \ldots \}, H \rangle \rightarrow \langle S[l \mapsto (d_1, \text{true})]; d_1 \rangle \).

  To show: \( \langle U_S(S); \text{freeze } l \text{ after } Q \text{ with } \lambda x. e_0, \{v, \ldots \}, H \rangle \rightarrow^i \text{error} \), where \( i' \leq 1 \).

  Consider whether \( U_S(S) = \top \):

  - If \( U_S(S) = \top \):

      In this case, by the definition of \text{error}, \( \langle U_S(S); \text{freeze } l \rangle = \text{error} \).
A. PROOFS

Hence \(\langle U_S(S); \text{freeze} l \rangle \xrightarrow{\iota'} \text{error}\), with \(i' = 0\).

- If \(U_S(S) \neq \top\):

Since \(U_S(S) \neq \top\) and \(S[l \mapsto (d_1, \text{true})] \neq \top\), but \(U_S(S[l \mapsto (d_1, \text{true})]) = \top\), it must be \(U_S\)'s action on the contents of \(l\) in \(S[l \mapsto (d_1, \text{true})]\) that updates \(S[l \mapsto (d_1, \text{true})]\) to \(\top\). Since the contents of \(l\) in \(S[l \mapsto (d_1, \text{true})]\) are \((d_1, \text{true})\), it must not be the case that \(S(l) = (d_1, \text{true})\), because otherwise, \(U_S(S)\) would be \(\top\). Therefore \(S(l) = (d_1, \text{false})\).

Let \(u_{p_1}\) be the state update operation in \(U_S\) that updates the contents of \(l\). Hence \(u_{p_1}((d_1, \text{true})) = \top\).

Recall that \(U_S\) is freeze-safe with \(\langle S; \text{freeze} l \text{ after } Q \text{ with } \lambda x. e_0, \{v, \ldots \}, H \rangle \xrightarrow{\iota} \langle S[l \mapsto (d_1, \text{true})]; d_1 \rangle\). By Definition 3.14, then, since the contents of \(l\) change in status during the transition from \(\langle S; \text{freeze} l \text{ after } Q \text{ with } \lambda x. e_0, \{v, \ldots \}, H \rangle\) to \(\langle S[l \mapsto (d_1, \text{true})]; d_1 \rangle\), we know that \(U_S\) either freezes the contents of \(l\) (having no other effect on them), or it acts as the identity on the contents of \(l\). Hence \((U_S(S[l \mapsto (d_1, \text{true}))))(l) = (d_1, \text{true})\). But this is a contradiction since \((U_S(S[l \mapsto (d_1, \text{true}))))(l) = u_{p_1}((d_1, \text{true})) = \top\).

Hence this case cannot occur.

LK: Is this argument convincing?

- Case E-Freeze-Simple:

Given: \(\langle S; \text{freeze} l \rangle \xrightarrow{\iota} \langle S[l \mapsto (d_1, \text{true})]; d_1 \rangle\).

To show: \(\langle U_S(S); \text{freeze} l \rangle \xrightarrow{\iota'} \text{error}\), where \(i' \leq 1\).

Consider whether \(U_S(S) = \top\):

- If \(U_S(S) = \top\):

In this case, by the definition of error, \(\langle U_S(S); \text{freeze} l \rangle = \text{error}\).

Hence \(\langle U_S(S); \text{freeze} l \rangle \xrightarrow{\iota'} \text{error}\), with \(i' = 0\).

- If \(U_S(S) \neq \top\):

Since \(U_S(S) \neq \top\) and \(S[l \mapsto (d_1, \text{true})] \neq \top\), but \(U_S(S[l \mapsto (d_1, \text{true})]) = \top\), it must be \(U_S\)'s action on the contents of \(l\) in \(S[l \mapsto (d_1, \text{true})]\) that updates \(S[l \mapsto (d_1, \text{true})]\) to
Since the contents of \( l \) in \( S[l \mapsto (d_1, \text{true})] \) are \((d_1, \text{true})\), it must not be the case that \( S(l) = (d_1, \text{true}) \), because otherwise, \( U_S(S) \) would be \( \top_S \). Therefore \( S(l) = (d_1, \text{false}) \).

Let \( u_{p_l} \) be the state update operation in \( U_S \) that updates the contents of \( l \). Hence \( u_{p_l}((d_1, \text{true})) = \top_p \).

Recall that \( U_S \) is freeze-safe with \( \langle S; \text{freeze } l \rangle \leftarrow \langle S[l \mapsto (d_1, \text{true})]; d_1 \rangle \). By Definition 3.14, then, since the contents of \( l \) change in status during the transition from \( \langle S; \text{freeze } l \rangle \) to \( \langle S[l \mapsto (d_1, \text{true})]; d_1 \rangle \), we know that \( U_S \) either freezes the contents of \( l \) (having no other effect on them), or it acts as the identity on the contents of \( l \). Hence \( (U_S(S[l \mapsto (d_1, \text{true}))))(l) = (d_1, \text{true}) \). But this is a contradiction since \( (U_S(S[l \mapsto (d_1, \text{true}))))(l) = u_{p_l}((d_1, \text{true})) = \top_p \). Hence this case cannot occur.

LK: Is this argument convincing? (It’s basically the same as that in the case above.)

A.17. Proof of Lemma 3.10

Proof. Suppose \( \sigma \mapsto \sigma_a \) and \( \sigma \mapsto \sigma_b \). We have to show that either there exist \( \sigma_c, i, j, \pi \) such that \( \sigma_a \mapsto^i \sigma_c \) and \( \pi(\sigma_b) \mapsto^j \sigma_c \) and \( i \leq 1 \) and \( j \leq 1 \), or that \( \sigma_a \mapsto \text{error} \) or \( \sigma_b \mapsto \text{error} \).

By inspection of the operational semantics, it must be the case that \( \sigma \) steps to \( \sigma_a \) 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 \( \sigma \) steps to \( \sigma_b \) 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.
A. PROOFS

First, consider the possibility that $E_a = E_b$ (and $e_{a_1} = e_{b_1}$). Since $\langle S; E_a[e_{a_1}] \rangle \rightarrow \langle S_a; E_a[e_{a_2}] \rangle$ by E-Eval-Ctxt and $\langle S; E_b[e_{b_1}] \rangle \rightarrow \langle S_b; E_b[e_{b_2}] \rangle$ by E-Eval-Ctxt, we have from the premise of E-Eval-Ctxt that $\langle S; e_{a_1} \rangle \rightarrow \langle S_a; e_{a_2} \rangle$ and $\langle S; e_{b_1} \rangle \rightarrow \langle S_b; e_{b_2} \rangle$. But then, since $e_{a_1} = e_{b_1}$, by Internal Determinism (Lemma 3.4) there is a permutation $\pi'$ such that $\langle S_a; e_{a_2} \rangle = \pi'(\langle S_b; e_{b_2} \rangle)$. Then we can satisfy the proof by choosing $\sigma_c = \langle S_a; e_{a_2} \rangle$ and $i = 0$ and $j = 0$ and $\pi = \pi'$.

The rest of this proof deals with the more interesting case in which $E_a \neq E_b$ (and $e_{a_1} \neq e_{b_1}$). Since $\langle S; E_a[e_{a_1}] \rangle \rightarrow \langle S_a; E_a[e_{a_2}] \rangle$ and $\langle S; E_b[e_{b_1}] \rangle \rightarrow \langle S_b; E_b[e_{b_2}] \rangle$ and $E_a[e_{a_1}] = E_b[e_{b_1}]$, where $E_a \neq E_b$, we have from Lemma 3.5 (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 $\sigma_c = \textbf{error}$, and in some we will prove that one of $\sigma_a$ or $\sigma_b$ steps to $\textbf{error}$. In most cases, however, our approach will be to show that there exist $S', i, j, \pi$ such that:

- $\langle S_a; E_a[e_{a_2}] \rangle \rightarrow^i \langle S'; E'_a[e_{a_2}] \rangle$, and
- $\pi((S_b; E_b[e_{b_2}])) \rightarrow^j \langle S'; E'_a[e_{a_2}] \rangle$.

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:

- $\langle S_a; E'_b[e_{b_1}] \rangle \rightarrow^i \langle S'; E'_b[e_{b_2}] \rangle$, and
- $\pi((S_b; E'_a[e_{a_1}])) \rightarrow^j \langle S'; E'_a[e_{a_2}] \rangle$.

From the premise of E-Eval-Ctxt, we have that $\langle S; e_{a_1} \rangle \rightarrow \langle S_a; e_{a_2} \rangle$ and $\langle S; e_{b_1} \rangle \rightarrow \langle S_b; e_{b_2} \rangle$. We proceed by case analysis on the rule by which $\langle S; e_{a_1} \rangle$ steps to $\langle S_a; e_{a_2} \rangle$. Since the only way an $\textbf{error}$ configuration can arise is by the E-Put-Err rule, we can assume in all other cases that $\sigma_a \neq \textbf{error}$.

1. Case E-Beta: We have $S_a = S$.  

DRAFT: March 14, 2015 205
We proceed by case analysis on the rule by which $\langle S; e_b \rangle$ steps to $\langle S_b; e_b \rangle$. Since the only way an error configuration can arise is by the E-Put-Err rule, we can assume in all other cases that $\sigma_b \neq \text{error}$.

(a) Case E-Beta: We have $S_a = S$ and $S_b = S$.

Choose $S' = S = S_a = S_b$, $i = 1$, $j = 1$, and $\pi = \text{id}$.

We have to show that:

- $\langle S; E'_b[e_{b_1}] \rangle \rightarrow \langle S_a; E'_b[e_{b_2}] \rangle$, and

- $\langle S; E'_a[e_{a_1}] \rangle \rightarrow \langle S_b; E'_a[e_{a_2}] \rangle$,

both of which follow immediately from $\langle S; e_{a_1} \rangle \rightarrow \langle S_a; e_{a_2} \rangle$ and $\langle S; e_{b_1} \rangle \rightarrow \langle S_b; e_{b_2} \rangle$ and E-Eval-Ctxt.

(b) Case E-New: We have $S_a = S$ and $S_b = S[l \mapsto (\bot, \text{false})]$.

Choose $S' = S_b$, $i = 1$, $j = 1$, and $\pi = \text{id}$.

We have to show that:

- $\langle S; E'_b[e_{b_1}] \rangle \rightarrow \langle S_b; E'_b[e_{b_2}] \rangle$, and

- $\langle S_b; E'_a[e_{a_1}] \rangle \rightarrow \langle S_b; E'_a[e_{a_2}] \rangle$.

The first of these follows immediately from $\langle S; e_{b_1} \rangle \rightarrow \langle S_b; e_{b_2} \rangle$ and E-Eval-Ctxt. For the second, consider that $S_b = S[l \mapsto (\bot, \text{false})] = U_S(S)$, where $U_S$ is the store update operation that acts as the identity on the contents of all existing locations, and adds the binding $l \mapsto (\bot, \text{false})$ if no binding for $l$ exists. Note that:

- $U_S$ is non-conflicting with $\langle S; e_{a_1} \rangle \rightarrow \langle S_a; e_{a_2} \rangle$, since no locations are allocated during the transition;

- $U_S(S_a) \neq \top_S$, since $U_S(S_a) = U_S(S) = S_b$ and we know that $\sigma_b \neq \text{error}$; and

- $U_S$ is freeze-safe with $\langle S; e_{a_1} \rangle \rightarrow \langle S_a; e_{a_2} \rangle$, since $S_a = S$, so there are no locations whose contents differ in status between them.
Therefore, by Lemma 3.7 (Generalized Independence), we have that \( \langle U_S(S); e_{a_1} \rangle \rightarrow \langle U_S(S_a); e_{a_2} \rangle \). Hence \( \langle S_b; e_{a_1} \rangle \rightarrow \langle S_b; e_{a_2} \rangle \). By E-Eval-Ctxt, it follows that \( \langle S_b; E'_a[e_{a_1}] \rangle \rightarrow \langle S_b; E'_a[e_{a_2}] \rangle \), as we were required to show.

(c) Case E-Put: We have \( S_a = S \) and \( S_b = S[l \mapsto u_{p_i}(p_1)] \).

Choose \( S' = S_b, i = 1, j = 1 \), and \( \pi = \text{id} \).

We have to show that:
- \( \langle S; E'_b[e_{b_1}] \rangle \rightarrow \langle S_b; E'_b[e_{b_2}] \rangle \), and
- \( \langle S_b; E'_a[e_{a_1}] \rangle \rightarrow \langle S_b; E'_a[e_{a_2}] \rangle \).

The first of these follows immediately from \( \langle S; e_{b_1} \rangle \rightarrow \langle S_b; e_{b_2} \rangle \) and E-Eval-Ctxt. For the second, consider that \( S_b = U_S(S) \), where \( U_S \) is the store update operation that applies \( u_{p_i} \) to the contents of \( l \) and acts as the identity on all other locations. Note that:
- \( U_S \) is non-conflicting with \( \langle S; e_{a_1} \rangle \rightarrow \langle S_a; e_{a_2} \rangle \), since no locations are allocated during the transition;
- \( U_S(S_a) \neq T_S \), since \( U_S(S_a) = U_S(S) = S_b \) and we know that \( \sigma_b \neq \text{error} \); and
- \( U_S \) is freeze-safe with \( \langle S; e_{a_1} \rangle \rightarrow \langle S_a; e_{a_2} \rangle \), since \( S_a = S \), so there are no locations whose contents differ in status between them.

Therefore, by Lemma 3.7 (Generalized Independence), we have that \( \langle U_S(S); e_{a_1} \rangle \rightarrow \langle U_S(S_a); e_{a_2} \rangle \). Hence \( \langle S_b; e_{a_1} \rangle \rightarrow \langle S_b; e_{a_2} \rangle \). By E-Eval-Ctxt, it follows that \( \langle S_b; E'_a[e_{a_1}] \rangle \rightarrow \langle S_b; E'_a[e_{a_2}] \rangle \), as we were required to show.

(d) Case E-Put-Err: We have \( S_a = S \) and \( \langle S_b; e_{b_2} \rangle = \text{error} \), and so we choose \( \sigma_c = \text{error}, i = 1, j = 0 \), and \( \pi = \text{id} \). We have to show that:
- \( \langle S; E'_b[e_{b_1}] \rangle \rightarrow \text{error} \), and
- \( \langle S_b; E'_a[e_{a_1}] \rangle = \text{error} \).

The second of these is immediately true because since \( \langle S_b; e_{b_2} \rangle = \text{error} \), \( S_b = T_S \), and so \( \langle S_b; E'_a[e_{a_1}] \rangle \) is equal to \text{error} as well. For the first, observe that \( \langle S; e_{b_1} \rangle \rightarrow \langle S_b; e_{b_2} \rangle \),
hence by E-Eval-Ctx, $\langle S; E'_b[e_{b_1}] \rangle \mapsto \langle S_b; E'_b[e_{b_2}] \rangle$. But $S_b = \top_S$, so $\langle S_b; E'_b[e_{b_2}] \rangle$ is equal to error, and so $\langle S; E'_b[e_{b_1}] \rangle \mapsto \text{error}$, as required.

(e) Case E-Get: Similar to case 1a, since $S_a = S$ and $S_b = S$.

(f) Case E-Freeze-Init: Similar to case 1a, since $S_a = S$ and $S_b = S$.

(g) Case E-Spawn-Handler: Similar to case 1a, since $S_a = S$ and $S_b = S$.

(h) Case E-Freeze-Final: We have $S_a = S$ and $S_b = S[l \mapsto (d_1, \text{true})]$.

Choose $S' = S_b$, $i = 1$, $j = 1$, and $\pi = \text{id}$.

We have to show that:

- $\langle S; E'_b[e_{b_1}] \rangle \mapsto \langle S_b; E'_b[e_{b_2}] \rangle$, and
- $\langle S_b; E'_a[e_{a_1}] \rangle \mapsto \langle S_b; E'_a[e_{a_2}] \rangle$.

The first of these follows immediately from $\langle S; e_{b_1} \rangle \mapsto \langle S_b; e_{b_2} \rangle$ and E-Eval-Ctx. For the second, note that $S_b = U_S(S)$, where $U_S$ is the store update operation that freezes the contents of $l$ and acts as the identity on the contents of all other locations. Note that:

- $U_S$ is non-conflicting with $\langle S; e_{a_1} \rangle \mapsto \langle S_a; e_{a_2} \rangle$, since no locations are allocated during the transition;
- $U_S(S_a) \neq \top_S$, since $U_S(S_a) = U_S(S) = S_b$ and we know that $\sigma_b \neq \text{error}$; and
- $U_S$ is freeze-safe with $\langle S; e_{a_1} \rangle \mapsto \langle S_a; e_{a_2} \rangle$, since $S_a = S$, so there are no locations whose contents differ in status between them.

Therefore, by Lemma 3.7 (Generalized Independence), we have that $\langle U_S(S); e_{a_1} \rangle \mapsto \langle U_S(S_a); e_{a_2} \rangle$.

Hence $\langle S_b; e_{a_1} \rangle \mapsto \langle S_b; e_{a_2} \rangle$. By E-Eval-Ctx, it follows that $\langle S_b; E'_a[e_{a_1}] \rangle \mapsto \langle S_b; E'_a[e_{a_2}] \rangle$, as we were required to show.

(i) Case E-Freeze-Simple: Similar to case 1h, since $S_b = S[l \mapsto (d_1, \text{true})]$.

(2) Case E-New: We have $S_a = S[l \mapsto (\bot, \text{false})]$.

We proceed by case analysis on the rule by which $\langle S; e_{b_1} \rangle$ steps to $\langle S_b; e_{b_2} \rangle$. Since the only way an error configuration can arise is by the E-Put-Err rule, we can assume in all other cases that $\sigma_b \neq \text{error}$. 

DRAFT: March 14, 2015 208
(a) Case E-Beta: By symmetry with case 1b.

(b) Case E-New: We have $S_a = S[l \mapsto (\bot, \text{false})]$ and $S_b = S[l' \mapsto (\bot, \text{false})]$. Now consider whether $l = l'$:

- If $l \neq l'$:

  Choose $S' = S[l' \mapsto (\bot, \text{false})][l \mapsto (\bot, \text{false})]$, $i = 1$, $j = 1$, and $\pi = \text{id}$.

  We have to show that:

  $$\langle S_a; E'_b[e_{b_1}] \rangle \longmapsto \langle S[l' \mapsto (\bot, \text{false})][l \mapsto (\bot, \text{false})]; E'_b[e_{b_2}] \rangle,$$

  and

  $$\langle S_b; E'_a[e_{a_1}] \rangle \longmapsto \langle S[l' \mapsto (\bot, \text{false})][l \mapsto (\bot, \text{false})]; E'_a[e_{a_2}] \rangle.$$

  For the first of these, consider that $S_a = S[l \mapsto (\bot, \text{false})] = U_S(S')$, where $U_S$ is the store update operation that acts as the identity on the contents of all existing locations, and adds the binding $l \mapsto (\bot, \text{false})$ if no binding for $l$ exists. Note that:

  - $U_S$ is non-conflicting with $\langle S; e_{b_1} \rangle \longmapsto \langle S_b; e_{b_2} \rangle$, since the only location allocated during the transition is $l'$, and $l \neq l'$ in this case;
  
  - $U_S(S_b) \neq \top_S$, since $U_S(S_b) = S[l' \mapsto (\bot, \text{false})][l \mapsto (\bot, \text{false})]$ and we know $S \neq \top_S$ and the addition of new bindings $l \mapsto (\bot, \text{false})$ and $l' \mapsto (\bot, \text{false})$ cannot cause it to become $\top_S$; and
  
  - $U_S$ is freeze-safe with $\langle S; e_{b_1} \rangle \longmapsto \langle S_b; e_{b_2} \rangle$, since $S_b = S[l' \mapsto (\bot, \text{false})]$ and $l' \notin \text{dom}(S)$, so there are no locations whose contents differ in status between $S$ and $S_b$.

  Therefore, by Lemma 3.7 (Generalized Independence), we have that $\langle U_S(S); e_{b_1} \rangle \longmapsto \langle U_S(S_b); e_{b_2} \rangle$. Hence $\langle S[l \mapsto (\bot, \text{false})]; e_{b_1} \rangle \longmapsto \langle S_b[l \mapsto (\bot, \text{false})]; e_{b_2} \rangle$. By E-Eval-Ctxt it follows that $\langle S[l \mapsto (\bot, \text{false})]; E'_b[e_{b_1}] \rangle \longmapsto \langle S_b[l \mapsto (\bot, \text{false})]; E'_b[e_{b_2}] \rangle$, which, since $S_b = S[l' \mapsto (\bot, \text{false})]$, 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 
\(E_a'[e_{a2}]\) (nor its equivalent, \(E_b'[e_{b2}]\)). The reason for this is that \(E_a'[e_{a2}]\) 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 \(l'' \notin \text{dom}(S)\) and \(l'' \neq l\) (and hence \(l'' \neq l'\), as well). Then choose \(S' = S[l'' \mapsto (\perp, \text{false})][l \mapsto (\perp, \text{false})], i = 1, j = 1,\) and \(\pi = \{(l, l'')\}\).

Either \(\langle S[l'' \mapsto (\perp, \text{false})][l \mapsto (\perp, \text{false})]; E_a'[\pi(e_{a2})]\rangle\) or \(\langle S[l'' \mapsto (\perp, \text{false})][l \mapsto (\perp, \text{false})]; E_b'[\pi(e_{b2})]\rangle\) would work as a final configuration; we choose \(\langle S[l'' \mapsto (\perp, \text{false})][l \mapsto (\perp, \text{false})]; E_b'[\pi(e_{b2})]\rangle\).

We have to show that:

- \(\langle S_a'; E_b'[e_{b1}]\rangle \mapsto \langle S[l'' \mapsto (\perp, \text{false})][l \mapsto (\perp, \text{false})]; E_b'[\pi(e_{b2})]\rangle\), and

- \(\pi(\langle S_b'; E_a'[e_{a1}]\rangle) \mapsto \langle S[l'' \mapsto (\perp, \text{false})][l \mapsto (\perp, \text{false})]; E_b'[\pi(e_{b2})]\rangle\).

For the first of these, since \(\langle S; e_{b1}\rangle \mapsto \langle S_b; e_{b2}\rangle\), we have by Lemma 3.3 (Permutability) that \(\pi(\langle S; e_{b1}\rangle) \mapsto \pi(\langle S_b; e_{b2}\rangle)\). Since \(\pi = \{(l, l'')\}, \) but \(l \notin S\) (from the side condition on E-New), we have that \(\pi(\langle S; e_{b1}\rangle) = \langle S; e_{b1}\rangle\). Since \(\langle S_b; e_{b2}\rangle = \langle S[l' \mapsto (\perp, \text{false})]; l'\rangle\), and \(l = l'\), we have that \(\pi(\langle S_b; e_{b2}\rangle) = \langle S[l'' \mapsto (\perp, \text{false})]; \pi(e_{b2})\rangle\).

Hence \(\langle S; e_{b1}\rangle \mapsto \langle S[l'' \mapsto (\perp, \text{false})]; \pi(e_{b2})\rangle\).

Let \(U_S\) be the store update operation that acts as the identity on the contents of all existing locations, and adds the binding \(l \mapsto (\perp, \text{false})\) if no binding for \(l\) exists. Note that:

- \(U_S\) is non-conflicting with \(\langle S; e_{b1}\rangle \mapsto \langle S[l'' \mapsto (\perp, \text{false})]; \pi(e_{b2})\rangle\), since the only location allocated during the transition is \(l''\);

- \(U_S(S[l'' \mapsto (\perp, \text{false})]) \neq \top_S\), since \(U_S(S[l'' \mapsto (\perp, \text{false})]) = S[l'' \mapsto (\perp, \text{false})][l \mapsto (\perp, \text{false})]\)

and \(l'' \mapsto (\perp, \text{false})\) cannot cause it to become \(\top_S\); and
A. PROOFS

- $U_S$ is freeze-safe with $\langle S; e_{b_1} \rangle \longmapsto \langle S[l'' \mapsto (\bot, \text{false})]; \pi(e_{b_1}) \rangle$, since $l'' \notin \text{dom}(S)$, so there are no locations whose contents differ in status between $S$ and $S[l'' \mapsto (\bot, \text{false})]$.

Therefore, by Lemma 3.7 (Generalized Independence), we have that $\langle U_S(S); e_{b_1} \rangle \longmapsto \langle U_S(S[l'' \mapsto (\bot, \text{false})]); \pi(e_{b_1}) \rangle$. Hence $\langle S[l \mapsto (\bot, \text{false})]; e_{b_1} \rangle \longmapsto \langle S[l'' \mapsto (\bot, \text{false})][l \mapsto (\bot, \text{false})]; \pi(e_{b_1}) \rangle$. By E-Eval-Ctx it follows that $\langle S[l \mapsto (\bot, \text{false})]; E'_b[\pi(e_{b_1})] \rangle \longmapsto \langle S[l'' \mapsto (\bot, \text{false})][l \mapsto (\bot, \text{false})]; E'_b[\pi(e_{b_1})] \rangle$, which, since $S[l \mapsto (\bot, \text{false})] = S_a$, is what we were required to show.

For the second, observe that since $S_b = S[l \mapsto (\bot, \text{false})]$, we have that $\pi(S_b) = S[l'' \mapsto (\bot, \text{false})]$. 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 $\langle S[l'' \mapsto (\bot, \text{false})]; (\pi(E'_a))[e_{a_1}] \rangle \longmapsto \langle S[l'' \mapsto (\bot, \text{false})][l \mapsto (\bot, \text{false})]; E'_b[\pi(e_{b_1})] \rangle$.

Let $U_S$ be the store update operation that acts as the identity on the contents of all existing locations, and adds the binding $l'' \mapsto (\bot, \text{false})$ if no binding for $l''$ exists. Note that:

- $U_S$ is non-conflicting with $\langle S; e_{a_1} \rangle \longmapsto \langle S_a; e_{a_2} \rangle$, since the only location allocated during the transition is $l$;
- $U_S(S_a) \neq \top_S$, since $U_S(S_a) = S[l'' \mapsto (\bot, \text{false})][l \mapsto (\bot, \text{false})]$ and we know $S \neq \top_S$ and the addition of new bindings $l \mapsto (\bot, \text{false})$ and $l'' \mapsto (\bot, \text{false})$ cannot cause it to become $\top_S$; and
- $U_S$ is freeze-safe with $\langle S; e_{a_1} \rangle \longmapsto \langle S_a; e_{a_2} \rangle$, since $S_a = S[l \mapsto (\bot, \text{false})]$ and $l \notin \text{dom}(S)$, so there are no locations whose contents differ in status between $S$ and $S_a$.

Therefore, by Lemma 3.7 (Generalized Independence), we have that $\langle U_S(S); e_{a_1} \rangle \longmapsto \langle U_S(S_a); e_{a_2} \rangle$. Hence $\langle S[l'' \mapsto (\bot, \text{false})]; e_{a_1} \rangle \longmapsto \langle S[l'' \mapsto (\bot, \text{false})][l \mapsto (\bot, \text{false})]; e_{a_2} \rangle$. By E-Eval-Ctx it follows that $\langle S[l'' \mapsto (\bot, \text{false})]; (\pi(E'_a))[e_{a_1}] \rangle \longmapsto \langle S[l'' \mapsto (\bot, \text{false})][l \mapsto (\bot, \text{false})]; E'_b[\pi(e_{b_1})] \rangle$. 

DRAFT: March 14, 2015
\[\langle S[l'' \mapsto (\bot, \text{false})][l \mapsto (\bot, \text{false})]; (\pi(E'_a))[e_{a_2}]\rangle,\] 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 \(S_a = S[l \mapsto (\bot, \text{false})]\) and \(S_b = S[l' \mapsto u_{p_i}(p_1)]\), where \(l \neq l'\) (since \(l \notin \text{dom}(S)\), but \(l' \in \text{dom}(S)\)).

We have to show that:

- \(\langle S_a; E'_b[e_{b_1}] \rangle \longmapsto \langle S_b[l \mapsto (\bot, \text{false})]; E'_b[e_{b_2}] \rangle\), and
- \(\langle S_b; E'_a[e_{a_1}] \rangle \longmapsto \langle S_b[l \mapsto (\bot, \text{false})]; E'_a[e_{a_2}] \rangle\).

For the first of these, consider that \(S_a = S[l \mapsto (\bot, \text{false})] = U_S(S)\), where \(U_S\) is the store update operation that acts as the identity on the contents of all existing locations, and adds the binding \(l \mapsto (\bot, \text{false})\) if no binding for \(l\) exists. Note that:

- \(U_S\) is non-conflicting with \(\langle S; e_{b_1} \rangle \longmapsto \langle S_b; e_{b_2} \rangle\), since no locations are allocated during the transition;
- \(U_S(S_b) \neq \top_S\), since \(U_S(S_b) = S_b[l \mapsto (\bot, \text{false})]\), and we know \(S_b \neq \top_S\) and the addition of a new binding \(l \mapsto (\bot, \text{false})\) cannot cause it to become \(\top_S\); and
- \(U_S\) is freeze-safe with \(\langle S; e_{b_1} \rangle \longmapsto \langle S_b; e_{b_2} \rangle\), since \(S_b = S[l' \mapsto u_{p_i}(p_1)]\) and \(u_{p_i}\) does not alter the status of \(p_1\). (By Definition 3.4, \(u_{p_i}\) can only change the status bit of a location if its contents are \((d, \text{true})\) and \(u_i(d) \neq d\), in which case \(u_{p_i}\) changes the contents of the location to \((\top, \text{false})\); however, that cannot be the case here since then \(u_{p_i}(p_1)\) would be \(\top_p\), contradicting the premise of E-Put.)

Therefore, by Lemma 3.7 (Generalized Independence), we have that \(\langle U_S(S); e_{b_1} \rangle \longmapsto \langle U_S(S_b); e_{b_2} \rangle\).

Hence \(\langle S[l \mapsto (\bot, \text{false})]; e_{b_1} \rangle \longmapsto \langle S_b[l \mapsto (\bot, \text{false})]; e_{b_2} \rangle\). By E-Eval-Ctxt, it follows that \(\langle S[l \mapsto (\bot, \text{false})]; E'_b[e_{b_1}] \rangle \longmapsto \langle S_b[l \mapsto (\bot, \text{false})]; E'_b[e_{b_2}] \rangle\), which, since \(S_a = S[l \mapsto (\bot, \text{false})]\), is what we were required to show.

For the second, let \(U_S\) be the store update operation that applies \(u_{p_i}\) to the contents of \(l'\) if it exists, and adds a binding \(l' \mapsto u_{p_i}(p_1)\) if no binding for \(l'\) exists.
A. PROOFS

Consider that $S_b = U_S(S)$, and $S_b[l \mapsto (\perp, \text{false})] = S_a[l' \mapsto u_{p_1}(p_1)] = U_S(S_a)$. Note that:

- $U_S$ is non-conflicting with $\langle S; e_{a_1} \rangle \longmapsto \langle S_a; e_{a_2} \rangle$, since the only location allocated during the transition is $l$;
- $U_S(S_a) \neq \top_S$, since $U_S(S_a) = S[l \mapsto (\perp, \text{false})][l' \mapsto u_{p_1}(p_1)]$ and we know $S \neq \top_S$ and the addition of new bindings $l \mapsto (\perp, \text{false})$ and $l' \mapsto u_{p_1}(p_1)$ cannot cause it to become $\top_S$ (since if $u_{p_1}(p_1) = \top_p$, $\langle S; e_{b_1} \rangle$ would not have been able to step by E-Put);
- $U_S$ is freeze-safe with $\langle S; e_{a_1} \rangle \longmapsto \langle S_a; e_{a_2} \rangle$, since $S_a = S[l \mapsto (\perp, \text{false})]$ and $l \notin \text{dom}(S)$, so there are no locations whose contents differ in status between $S$ and $S_a$.

Therefore, by Lemma 3.7 (Generalized Independence), we have that $\langle U_S(S); e_{a_1} \rangle \longmapsto \langle U_S(S_a); e_{a_2} \rangle$. Hence $\langle S_b; e_{a_1} \rangle \longmapsto \langle S_b[l \mapsto (\perp, \text{false})]; e_{a_2} \rangle$. By E-Eval-Ctxt, it follows that $\langle S_b; E'_a[e_{a_1}] \rangle \longmapsto \langle S_b[l \mapsto (\perp, \text{false})]; E'_a[e_{a_2}] \rangle$, as we were required to show.

(d) Case E-Put-Err: We have $S_a = S[l \mapsto (\perp, \text{false})]$ and $\langle S_b; e_{b_2} \rangle = \text{error}$, and so we choose $\sigma_c = \text{error}$, $i = 1$, $j = 0$, and $\pi = \text{id}$. We have to show that:

- $\langle S_a; E'_b[e_{b_1}] \rangle \longmapsto \text{error}$, and
- $\langle S_b; E'_a[e_{a_1}] \rangle = \text{error}$.

The second of these is immediately true because since $\langle S_b; e_{b_2} \rangle = \text{error}$, $S_b = \top_S$, and so $\langle S_b; E'_a[e_{a_1}] \rangle$ is equal to $\text{error}$ as well. For the first, observe that since $\langle S; e_{a_1} \rangle \longmapsto \langle S_a; e_{a_2} \rangle$, we have by Lemma 3.6 (Monotonicity) that $S \sqsubseteq_S S_a$. Therefore, since $\langle S; e_{b_1} \rangle \longmapsto \text{error}$, we have by Lemma 3.9 (Error Preservation) that $\langle S_a; e_{b_1} \rangle \longmapsto \text{error}$. Since $\text{error}$ is equal to $\langle \top_S; e \rangle$ for all expressions $e$, $\langle S_a; e_{b_1} \rangle \longmapsto \langle \top_S; e \rangle$ 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 $\text{error}$, we have that $\langle S_a; E'_b[e_{b_1}] \rangle \longmapsto \text{error}$, as we were required to show.

(e) Case E-Get: Similar to case 2a, since $S_a = S[l \mapsto (\perp, \text{false})]$ and $S_b = S$. 

DRAFT: March 14, 2015
(f) Case E-Freeze-Init: Similar to case 2a, since $S_a = S[l \mapsto (\bot, \text{false})]$ and $S_b = S$.

(g) Case E-Spawn-Handler: Similar to case 2a, since $S_a = S[l \mapsto (\bot, \text{false})]$ and $S_b = S$.

(h) Case E-Freeze-Final: We have $S_a = S[l \mapsto (\bot, \text{false})]$ and $S_b = S[l' \mapsto (d_1, \text{true})]$, where $l \neq l'$ (since $l \notin \text{dom}(S)$, but $l' \in \text{dom}(S)$).

Choose $S' = S[l \mapsto (\bot, \text{false})][l' \mapsto (d_1, \text{true})]$, $i = j = 1$, and $\pi = \text{id}$.

We have to show that:

- $\langle S[l \mapsto (\bot, \text{false})]; E_a^b[e_{b_1}] \rangle \leftrightarrow \langle S[l \mapsto (\bot, \text{false})][l' \mapsto (d_1, \text{true})]; E_a^b[e_{b_2}] \rangle$, and
- $\langle S[l' \mapsto (d_1, \text{true})]; E_a^e[e_{a_1}] \rangle \leftrightarrow \langle S[l \mapsto (\bot, \text{false})][l' \mapsto (d_1, \text{true})]; E_a^e[e_{a_2}] \rangle$.

For the first of these, consider that $S[l \mapsto (\bot, \text{false})] = U_S(S)$, where $U_S$ is the store update operation that acts as the identity on the contents of all existing locations, and adds the binding $l \mapsto (\bot, \text{false})$ if no binding for $l$ exists. Note that:

- $U_S$ is non-conflicting with $\langle S; e_{b_1} \rangle \leftrightarrow \langle S_b; e_{b_2} \rangle$, since no locations are allocated during the transition;
- $U_S(S_b) \neq T_S$, since $U_S(S_b) = S_b[l \mapsto (\bot, \text{false})]$, and we know $S_b \neq T_S$ and the addition of a new binding $l \mapsto (\bot, \text{false})$ cannot cause it to become $T_S$; and
- $U_S$ is freeze-safe with $\langle S; e_{b_1} \rangle \leftrightarrow \langle S_b; e_{b_2} \rangle$, since $S_b = S[l' \mapsto (d_1, \text{true})]$ and so the only location that can change in status between $S$ and $S_b$ is $l'$, and $U_S$ acts as the identity on $l'$.

Therefore, by Lemma 3.7 (Generalized Independence), we have that $\langle U_S(S); e_{b_1} \rangle \leftrightarrow \langle U_S(S_b); e_{b_2} \rangle$.

Hence $\langle S[l \mapsto (\bot, \text{false})]; e_{b_1} \rangle \leftrightarrow \langle S[l \mapsto (\bot, \text{false})][l' \mapsto (d_1, \text{true})]; e_{b_2} \rangle$. By E-Eval-Ctx, it follows that $\langle S[l \mapsto (\bot, \text{false})]; E_a^b[e_{b_1}] \rangle \leftrightarrow \langle S[l \mapsto (\bot, \text{false})][l' \mapsto (d_1, \text{true})]; E_a^b[e_{b_2}] \rangle$, as we were required to show.

For the second, consider that $S[l' \mapsto (d_1, \text{true})] = U_S(S)$, where $U_S$ is the store update operation that freezes the contents of $l'$ and acts as the identity on the contents of all other locations. Note that:
A. PROOFS

- $U_S$ is non-conflicting with $(S; e_{a_1}) \rightarrow (S_a; e_{a_2})$, since the only location allocated during the transition is $l$, and $l \neq l'$;
- $U_S(S_a) \neq \top_S$, since $U_S(S_a) = S_a[l' \mapsto (d_1, \text{true})] = S_b[l \mapsto (\bot, \text{false})]$, and we know $S_b \neq \top_S$ and the addition of a new binding $l \mapsto (\bot, \text{false})$ cannot cause it to become $\top_S$; and
- $U_S$ is freeze-safe with $(S; e_{a_1}) \rightarrow (S_a; e_{a_2})$, since $S_a = S[l \mapsto (\bot, \text{false})]$ and $l \notin \text{dom}(S)$, so there are no locations whose contents differ in status between $S$ and $S_a$.

Therefore, by Lemma 3.7 (Generalized Independence), we have that $(U_S(S); e_{a_1}) \rightarrow (U_S(S_a); e_{a_2})$.

Hence $(S[l' \mapsto (d_1, \text{true})]; e_{a_1}) \rightarrow (S[l \mapsto (\bot, \text{false})][l' \mapsto (d_1, \text{true})]; e_{a_2})$. By E-Eval-Ctxt it follows that $(S[l' \mapsto (d_1, \text{true})]; E'_a[e_{a_1}]) \rightarrow (S[l \mapsto (\bot, \text{false})][l' \mapsto (d_1, \text{true})]; E'_a[e_{a_2}])$, as we were required to show.

(i) Case E-Freeze-Simple: Similar to case 2h, since $S_a = S[l \mapsto (\bot, \text{false})]$ and $S_b = S[l' \mapsto (d_1, \text{true})]$, where $l \neq l'$ (since $l \notin \text{dom}(S)$, but $l' \in \text{dom}(S)$).

(3) Case E-Put: We have $S_a = S[l \mapsto u_{p_i}(p_1)]$.

We proceed by case analysis on the rule by which $(S; e_{b_1})$ steps to $(S_b; e_{b_2})$. Since the only way an {\textbf{error}} configuration can arise is by the E-Put-Err rule, we can assume in all other cases that $\sigma_b \neq \text{error}$.

(a) Case E-Beta: By symmetry with case 1c.

(b) Case E-New: By symmetry with case 2c.

(c) Case E-Put: We have $S_a = S[l \mapsto u_{p_i}(p_1)]$ and $S_b = S[l' \mapsto u_{p_j}(p'_1)]$, where $p'_1 = S(l')$.

Now consider whether $l = l'$:

- If $l \neq l'$:

  Choose $S' = S[l' \mapsto u_{p_j}(p'_1)][l \mapsto u_{p_i}(p_1)]$, $i = 1$, $j = 1$, and $\pi = \text{id}$.

  We have to show that:

  - $(S[l \mapsto u_{p_i}(p_1)]; E'_b[e_{b_1}]) \rightarrow (S[l' \mapsto u_{p_j}(p'_1)][l \mapsto u_{p_i}(p_1)]; E'_b[e_{b_2}])$, and
- \( \langle S[l' \mapsto u_{p_j}(p'_1)]; E'_a[e_{a_1}] \rangle \mapsto \langle S[l' \mapsto u_{p_j}(p'_1)]l \mapsto u_{p_i}(p_1)]; E'_a[e_{a_2}] \rangle \).

For the first of these, consider that \( S[l \mapsto u_{p_i}(p_1)] = U_S(S) \), where \( U_S \) is the store update operation that applies \( u_{p_i} \) to the contents of \( l \) if it exists, and adds a binding \( l \mapsto u_{p_i}(p_1) \) if no binding for \( l \) exists. Note that:

- \( U_S \) is non-conflicting with \( \langle S; e_{b_1} \rangle \mapsto \langle S[l' \mapsto u_{p_j}(p'_1)]; e_{b_2} \rangle \), since no locations are allocated during the transition;

- \( U_S(S[l' \mapsto u_{p_j}(p'_1)]) \neq \top_S \), since \( U_S(S[l' \mapsto u_{p_j}(p'_1)]) = S[l' \mapsto u_{p_j}(p'_1)]l \mapsto u_{p_i}(p_1) \) and we know \( S \neq \top_S \) and the addition of new bindings \( l \mapsto u_{p_i}(p_1) \) and \( l' \mapsto u_{p_j}(p'_1) \) cannot cause it to become \( \top_S \) (because if one of them could, then we would have \( S_a = \text{error} \) or \( S_b = \text{error} \), which we know are not the case); and

- \( U_S \) is freeze-safe with \( \langle S; e_{b_1} \rangle \mapsto \langle S[l' \mapsto u_{p_j}(p'_1)]; e_{b_2} \rangle \), since \( u_{p_j} \) does not alter the status of \( p'_1 \). (By Definition 3.4, \( u_{p_j} \) can only change the status bit of a location if its contents are \((d, \text{true})\) and \( u_j(d) \neq d \), in which case \( u_{p_j} \) changes the contents of the location to \((\top, \text{false})\); however, that cannot be the case here since then \( u_{p_j}(p'_1) \) would be \( \top_p \), contradicting the premise of E-Put.)

Therefore, by Lemma 3.7 (Generalized Independence), we have that \( \langle U_S(S); e_{b_1} \rangle \mapsto \langle U_S(S[l' \mapsto u_{p_j}(p'_1)]); e_{b_2} \rangle \). Hence \( \langle S[l \mapsto u_{p_i}(p_1)]; e_{b_1} \rangle \mapsto \langle S[l' \mapsto u_{p_j}(p'_1)]l \mapsto u_{p_i}(p_1)]; e_{b_2} \rangle \). By E-Eval-Ctxt, it follows that \( \langle S[l \mapsto u_{p_i}(p_1)]; E'_b[e_{b_1}] \rangle \mapsto \langle S[l' \mapsto u_{p_j}(p'_1)][l \mapsto u_{p_i}(p_1)]; E'_b[e_{b_2}] \rangle \), as we were required to show. The argument for the second is symmetrical.

- If \( l = l' \): Note that since \( l = l', p_1 = p'_1 \) as well.

Consider whether \( u_{p_i}(u_{p_j}(p_1)) = \top_p \):

- If \( u_{p_i}(u_{p_j}(p_1)) = \top_p \):

  Choose \( \sigma_c = \text{error}, i = 1, j = 1, \) and \( \pi = \text{id} \). We have to show that:

  * \( \langle S[l \mapsto u_{p_i}(p_1)]; E'_b[e_{b_1}] \rangle \mapsto \text{error} \), and
  *

  * \( \langle S[l \mapsto u_{p_j}(p_1)]; E'_a[e_{a_1}] \rangle \mapsto \text{error} \).
A. PROOFS

For the first of these, consider that $S[l \mapsto u_{p_1}(p_1)] = U_S(S)$, where $U_S$ is the store update operation that applies $u_{p_1}$ to the contents of $l$ if it exists. Note that:

* $U_S$ is non-conflicting with $S; e_{b_1} \longmapsto S[l \mapsto u_{p_1}(p_1)]; e_{b_2}$, since no locations are allocated during the transition;

* $U_S(S[l \mapsto u_{p_1}(p_1)]) = T_S$, since $U_S(S[l \mapsto u_{p_1}(p_1)]) = S[l \mapsto u_{p_1}(u_{p_1}(p_1))]$ and we know $u_{p_1}(u_{p_1}(p_1)) = T_p$ in this case;

* $U_S$ is freeze-safe with $S; e_{b_1} \longmapsto S[l \mapsto u_{p_1}(p_1)]; e_{b_2}$, since $u_{p_1}$ does not alter the status of $p_1$. (By Definition 3.4, $u_{p_1}$ can only change the status bit of a location if its contents are $(d, \text{true})$ and $u_{j}(d) \neq d$, in which case $u_{p_1}$ changes the contents of the location to $(\top, \text{false})$; however, that cannot be the case here since then $u_{p_1}(p_1)$ would be $T_p$, contradicting the premise of E-Put.)

Therefore, by Lemma 3.8 (Generalized Clash), we have that there exists $i'' \leq 1$ such that $U_S(S); e_{b_1} \longmapsto i'' \text{error}$. Hence $S[l \mapsto u_{p_1}(p_1)]; e_{b_1} \longmapsto i'' \text{error}$.

If $i'' = 0$, we would have $S[l \mapsto u_{p_1}(p_1)]; e_{b_1} = S_a; e_{b_1} = \text{error}$. So we would have $S_a = T_S$ by the definition of \text{error}, but then we would have $\sigma_a = \text{error}$, a contradiction. Therefore $i'' = 1$, and so we have $S[l \mapsto u_{p_1}(p_1)]; e_{b_1} \longmapsto \text{error}$.

Since $\text{error} = \langle T_S; e \rangle$ for all $e$, we have $S[l \mapsto u_{p_1}(p_1)]; e_{b_1} \longmapsto \langle T_S; e \rangle$ for all $e$. So, by E-Eval-Ctxt, we have that $S[l \mapsto u_{p_1}(p_1)]; E_b'[e_{b_1}] \longmapsto \langle T_S; E_b'[e] \rangle$ for all $e$. Hence $S[l \mapsto u_{p_1}(p_1)]; E_b'[e_{b_1}] \longmapsto \text{error}$. The argument for the second is symmetrical.

- If $u_{p_1}(u_{p_1}(p_1)) \neq T_p$:

  Choose $S' = S[l \mapsto u_{p_1}(u_{p_1}(p_1))], i = 1, j = 1, \text{ and } \pi = \text{id}$.

  We have to show that:

  * $\langle S[l \mapsto u_{p_1}(p_1)]; E_b'[e_{b_1}] \rangle \longmapsto \langle S[l \mapsto u_{p_1}(u_{p_1}(p_1))]; E_b'[e_{b_2}] \rangle$, and

  * $\langle S[l \mapsto u_{p_1}(p_1)]; E_a'[e_{a_1}] \rangle \longmapsto \langle S[l \mapsto u_{p_1}(u_{p_1}(p_1))]; E_a'[e_{a_2}] \rangle$.  

For the first of these, consider that $S[l \mapsto u_{p_i}(p_1)] = U_S(S)$, where $U_S$ is the store update operation that applies $u_{p_i}$ to the contents of $l$ if it exists. Note that:
* $U_S$ is non-conflicting with $\langle S; e_{b_1} \rangle \longmapsto \langle S[l \mapsto u_{p_j}(p_1)]; e_{b_2} \rangle$, since no locations are allocated during the transition;
* $U_S(S[l \mapsto u_{p_j}(p_1)]) \neq \top_S$, since $U_S(S[l \mapsto u_{p_j}(p_1)]) = S[l \mapsto u_{p_i}(u_{p_j}(p_1))]$ and we know $S \neq \top_S$ and $u_{p_i}(u_{p_j}(p_1)) \neq \top_p$ in this case;
* $U_S$ is freeze-safe with $\langle S; e_{b_1} \rangle \longmapsto \langle S[l \mapsto u_{p_j}(p_1)]; e_{b_2} \rangle$, since $u_{p_j}$ does not alter the status of $p_1$. (By Definition 3.4, $u_{p_j}$ can only change the status bit of a location if its contents are $(d, \text{true})$ and $u_j(d) \neq d$, in which case $u_{p_j}$ changes the contents of the location to $(\top, \text{false})$; however, that cannot be the case here since then $u_{p_j}(p_1)$ would be $\top_p$, contradicting the premise of E-Put.)

Therefore, by Lemma 3.7 (Generalized Independence), we have that $\langle U_S(S); e_{b_1} \rangle \longmapsto \langle U_S(S[l \mapsto u_{p_j}(p_1)]); e_{b_2} \rangle$. Hence $\langle S[l \mapsto u_{p_i}(p_1)]; e_{b_1} \rangle \longmapsto \langle S[l \mapsto u_{p_i}(u_{p_j}(p_1))]; e_{b_2} \rangle$.

By E-Eval-Ctxt, it follows that $\langle S[l \mapsto u_{p_i}(p_1)]; E_b'[e_{b_1}] \rangle \longmapsto \langle S[l \mapsto u_{p_i}(u_{p_j}(p_1))]; E_b'[e_{b_2}] \rangle$ as we were required to show. The argument for the second is symmetrical.

(d) Case E-Put-Err: We have $S_a = S[l \mapsto u_{p_i}(p_1)]$ and $\langle S_b; e_{b_2} \rangle = \text{error}$, and so we choose $\sigma_c = \text{error}$, $i = 1$, $j = 0$, and $\pi = \text{id}$. We have to show that:

- $\langle S_a; E_b'[e_{b_1}] \rangle \longmapsto \text{error}$, and
- $\langle S_b; E_a'[e_{a_1}] \rangle = \text{error}$.

The second of these is immediately true because since $\langle S_b; e_{b_2} \rangle = \text{error}$, $S_b = \top_S$, and so $\langle S_b; E_a'[e_{a_1}] \rangle$ is equal to $\text{error}$ as well. For the first, observe that since $\langle S; e_{a_1} \rangle \longmapsto \langle S_a; e_{a_2} \rangle$, we have by Lemma 3.6 (Monotonicity) that $S \sqsubseteq_S S_a$. Therefore, since $\langle S; e_{b_1} \rangle \longmapsto \text{error}$, we have by Lemma 3.9 (Error Preservation) that $\langle S_a; e_{b_1} \rangle \longmapsto \text{error}$. Since $\text{error}$ is equal to $\langle \top_S; e \rangle$ for all expressions $e$, $\langle S_a; e_{b_1} \rangle \longmapsto \langle \top_S; e \rangle$ 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 $\text{error}$, we have that $\langle S_a; E_b'[e_{b_1}] \rangle \longmapsto \text{error}$, as we were required to show.
(e) Case E-Get: Similar to case 3a, since $S_a = S[l \mapsto u_{p_i}(p_1)]$ and $S_b = S$.

(f) Case E-Freeze-Init: Similar to case 3a, since $S_a = S[l \mapsto u_{p_i}(p_1)]$ and $S_b = S$.

(g) Case E-Spawn-Handler: Similar to case 3a, since $S_a = S[l \mapsto u_{p_i}(p_1)]$ and $S_b = S$.

(h) Case E-Freeze-Final: We have $S_a = S[l \mapsto u_{p_i}(p_1)]$ and $S_b = S[l' \mapsto (d_1,\text{true})]$.

Now consider whether $l = l'$:

- If $l \not= l'$:

Choose $S' = S[l \mapsto u_{p_i}(p_1)][l' \mapsto (d_1,\text{true})], i = 1, j = 1, \text{and } \pi = \text{id}$.

We have to show that:

- $\langle S[l \mapsto u_{p_i}(p_1)]; E'_b[e_{b_1}] \rangle \longrightarrow \langle S[l \mapsto u_{p_i}(p_1)][l' \mapsto (d_1,\text{true})]; E'_b[e_{b_2}] \rangle$, and

- $\langle S[l' \mapsto (d_1,\text{true})]; E'_a[e_{a_1}] \rangle \longrightarrow \langle S[l \mapsto u_{p_i}(p_1)][l' \mapsto (d_1,\text{true})]; E'_a[e_{a_2}] \rangle$.

For the first of these, consider that $S[l \mapsto u_{p_i}(p_1)] = U_S(S), \text{where } U_S$ is the store update operation that applies $u_{p_i}$ to the contents of $l$ if it exists, and adds a binding $l \mapsto u_{p_i}(p_1)$ if no binding for $l$ exists, and acts as the identity on all other locations.

Note that:

- $U_S$ is non-conflicting with $\langle S; e_{b_1} \rangle \longrightarrow \langle S[l' \mapsto (d_1,\text{true})]; e_{b_2} \rangle$, since no locations are allocated during the transition;

- $U_S(S[l' \mapsto (d_1,\text{true})]) \not= T_S, \text{since } U_S(S[l' \mapsto (d_1,\text{true})]) = S[l' \mapsto (d_1,\text{true})][l \mapsto u_{p_i}(p_1)]$ and we know $S \not= T_S$ and the addition of new bindings $l \mapsto u_{p_i}(p_1)$ and $l' \mapsto (d_1,\text{true})$ cannot cause it to become $T_S$ (because if one of them could, then we would have $S_a = \text{error} \text{ or } S_b = \text{error}, \text{which we know are not the case});$ and

- $U_S$ is freeze-safe with $\langle S; e_{b_1} \rangle \longrightarrow \langle S[l' \mapsto (d_1,\text{true})]; e_{b_2} \rangle$, since the only location that can change in status between $S$ and $S[l' \mapsto (d_1,\text{true})]$ is $l'$, and $U_S$ acts as the identity on $l'$.

Therefore, by Lemma 3.7 (Generalized Independence), we have that $\langle U_S(S); e_{b_1} \rangle \longrightarrow \langle U_S(S[l' \mapsto (d_1,\text{true})]); e_{b_2} \rangle$. Hence $\langle S[l \mapsto u_{p_i}(p_1)]; e_{b_1} \rangle \longrightarrow \langle S[l' \mapsto (d_1,\text{true})][l}
\[ u_{p_1}(p_1); e_{b_2} \]. By E-Eval-Ctxt, it follows that \( \langle S[l \mapsto u_{p_1}(p_1)]; E'_b[e_{b_1}] \rangle \rightarrow \langle S[l' \mapsto (d_1, \text{true})][l \mapsto u_{p_1}(p_1)]; E'_b[e_{b_2}] \rangle \), as we were required to show.

For the second, consider that \( S[l' \mapsto (d_1, \text{true})] = U_S(S) \), where \( U_S \) is the store update operation that freezes the contents of \( l' \) and acts as the identity on the contents of all other locations. Note that:

- \( U_S \) is non-conflicting with \( \langle S; e_{a_1} \rangle \rightarrow \langle S[l \mapsto u_{p_1}(p_1)]; e_{a_2} \rangle \), since no locations are allocated during the transition;
- \( U_S(S[l \mapsto u_{p_1}(p_1)]) \neq \top_S \), since \( U_S(S[l \mapsto u_{p_1}(p_1)]) = S[l \mapsto u_{p_1}(p_1)][l' \mapsto (d_1, \text{true})] \), and we know \( S \neq \top_S \) and the addition of new bindings \( l \mapsto u_{p_1}(p_1) \) and \( l' \mapsto (d_1, \text{true}) \) cannot cause it to become \( \top_S \) (because if one of them could, then we would have \( S_a = \text{error} \) or \( S_b = \text{error} \), which we know are not the case);

and

- \( U_S \) is freeze-safe with \( \langle S; e_{a_1} \rangle \rightarrow \langle S[l \mapsto u_{p_1}(p_1)]; e_{a_2} \rangle \), since \( u_{p_1} \) does not alter the status of \( p_1 \). (By Definition 3.4, \( u_{p_1} \) can only change the status bit of a location if its contents are \( (d, \text{true}) \) and \( u_i(d) \neq d \), in which case \( u_{p_1} \) changes the contents of the location to \( (\top, \text{false}) \); however, that cannot be the case here since then \( u_{p_1}(p_1) \) would be \( \top_p \), and we would have \( S_a = \top_S \), a contradiction.)

Therefore, by Lemma 3.7 (Generalized Independence), we have that \( \langle U_S(S); e_{a_1} \rangle \rightarrow \langle U_S(S[l \mapsto u_{p_1}(p_1)]); e_{a_2} \rangle \). Hence \( \langle S[l' \mapsto (d_1, \text{true})]; e_{a_1} \rangle \rightarrow \langle S[l \mapsto u_{p_1}(p_1)][l' \mapsto (d_1, \text{true})]; e_{a_2} \rangle \). By E-Eval-Ctxt, it follows that \( \langle S[l' \mapsto (d_1, \text{true})]; E'_a[e_{a_1}] \rangle \rightarrow \langle S[l \mapsto u_{p_1}(p_1)][l' \mapsto (d_1, \text{true})]; E'_a[e_{a_2}] \rangle \), as we were required to show.

- If \( l = l' \):

Note that since \( l = l', p_1 = (d_1, \text{true}) \).

**TODO:** This is the one interesting case in the whole thing—the conflicting-put-and-freeze case.
A. PROOFS

(i) Case E-Freeze-Simple: Similar to case 3h, since $S_a = S[l \mapsto u_{p_i}(p_1)]$ and $S_b = S'[l' \mapsto (d_1, \text{true})]$.

(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 $\langle S_b; e_{b_2} \rangle$. Since the only way an error configuration can arise is by the E-Put-Err rule, we can assume in all other cases that $\sigma_b \neq \textbf{error}$.

(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: We have $\langle S_a; e_{a_2} \rangle = \textbf{error}$ and $\langle S_b; e_{b_2} \rangle = \textbf{error}$, 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 $\langle S_a; e_{a_2} \rangle = \textbf{error}$ and $S_b = S$.

(f) Case E-Freeze-Init: Similar to case 4a, since $\langle S_a; e_{a_2} \rangle = \textbf{error}$ and $S_b = S$.

(g) Case E-Spawn-Handler: Similar to case 4a, since $\langle S_a; e_{a_2} \rangle = \textbf{error}$ and $S_b = S$.

(h) Case E-Freeze-Final: We have $\langle S_a; e_{a_2} \rangle = \textbf{error}$ and $S_b = S[l \mapsto (d_1, \text{true})]$, and so we choose $\sigma_c = \textbf{error}$, $i = 0$, $j = 1$, and $\pi = \text{id}$. We have to show that:

- $\langle S_a; E'_b[e_{b_1}] \rangle = \textbf{error}$, and
- $\langle S_b; E'_a[e_{a_1}] \rangle \leftarrow \textbf{error}$.

The first of these is immediately true because since $\langle S_a; e_{a_2} \rangle = \textbf{error}$, $S_a = \top_S$, and so $\langle S_a; E'_b[e_{b_1}] \rangle$ is equal to $\textbf{error}$ as well. For the second, observe that since $\langle S; e_{b_1} \rangle \leftarrow \langle S_b; e_{b_2} \rangle$, we have by Lemma 3.6 (Monotonicity) that $S \sqsubseteq S_b$. Therefore, since $\langle S; e_{a_1} \rangle \leftarrow \textbf{error}$, we have by Lemma 3.9 that $\langle S_b; e_{a_1} \rangle \leftarrow \textbf{error}$. Since $\textbf{error}$ is equal to $\langle \top_S; e \rangle$ for all
expressions $e$, $\langle S_b; e_{a_1} \rangle \rightarrow \langle T_S; e \rangle$ for all $e$. Therefore, by E-Eval-Ctxt, $\langle S_b; E'_a[e_{a_1}] \rangle \rightarrow \langle T_S; E'_a[e] \rangle$ for all $e$. Since $\langle T_S; E'_a[e] \rangle$ is equal to $\text{error}$, we have that $\langle S_b; E'_a[e_{a_1}] \rangle \rightarrow \text{error}$, as we were required to show.

(i) Case E-Freeze-Simple: Similar to case 4h, since $S_b = S[l \mapsto (d_1, \text{true})]$.

(5) Case E-Get: We have $S_a = S$.

We proceed by case analysis on the rule by which $\langle S; e_{b_1} \rangle$ steps to $\langle S_b; e_{b_2} \rangle$. Since the only way an $\text{error}$ configuration can arise is by the E-Put-Err rule, we can assume in all other cases that $\sigma_b \neq \text{error}$.

(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 $S_a = S$ and $S_b = S$.

(f) Case E-Freeze-Init: Similar to case 5a, since $S_a = S$ and $S_b = S$.

(g) Case E-Spawn-Handler: Similar to case 5a, since $S_a = S$ and $S_b = S$.

(h) Case E-Freeze-Final: Similar to case 1h, since $S_a = S$ and $S_b = S[l \mapsto (d_1, \text{true})]$.

(i) Case E-Freeze-Simple: Similar to case 1i, since $S_a = S$ and $S_b = S[l \mapsto (d_1, \text{true})]$.

(6) Case E-Freeze-Init: We have $S_a = S$.

We proceed by case analysis on the rule by which $\langle S; e_{b_1} \rangle$ steps to $\langle S_b; e_{b_2} \rangle$. Since the only way an $\text{error}$ configuration can arise is by the E-Put-Err rule, we can assume in all other cases that $\sigma_b \neq \text{error}$.

(a) Case E-Beta: By symmetry with case 1f.

(b) Case E-New: By symmetry with case 2f.

(c) Case E-Put: By symmetry with case 3f.

(d) Case E-Put-Err: By symmetry with case 4f.

(e) Case E-Get: By symmetry with case 5f.
A. PROOFS

(f) Case E-Freeze-Init: Similar to case 6a, since $S_a = S$ and $S_b = S$.

(g) Case E-Spawn-Handler: Similar to case 6a, since $S_a = S$ and $S_b = S$.

(h) Case E-Freeze-Final: Similar to case 1h, since $S_a = S$ and $S_b = S[l \mapsto (d_1, \text{true})]$.

(i) Case E-Freeze-Simple: Similar to case 1i, since $S_a = S$ and $S_b = S[l \mapsto (d_1, \text{true})]$.

(7) Case E-Spawn-Handler: We have $S_a = S$.

We proceed by case analysis on the rule by which $\langle S; e_{b_1} \rangle$ steps to $\langle S_b; e_{b_2} \rangle$. Since the only way an error configuration can arise is by the E-Put-Err rule, we can assume in all other cases that $\sigma_b \neq \text{error}$.

(a) Case E-Beta: By symmetry with case 1g.

(b) Case E-New: By symmetry with case 2g.

(c) Case E-Put: By symmetry with case 3g.

(d) Case E-Put-Err: By symmetry with case 4g.

(e) Case E-Get: By symmetry with case 5g.

(f) Case E-Freeze-Init: By symmetry with case 6g.

(g) Case E-Spawn-Handler: Similar to case 7a, since $S_a = S$ and $S_b = S$.

(h) Case E-Freeze-Final: Similar to case 1h, since $S_a = S$ and $S_b = S[l \mapsto (d_1, \text{true})]$.

(i) Case E-Freeze-Simple: Similar to case 1i, since $S_a = S$ and $S_b = S[l \mapsto (d_1, \text{true})]$.

(8) Case E-Freeze-Final: We have $S_a = S[l \mapsto (d_1, \text{true})]$.

We proceed by case analysis on the rule by which $\langle S; e_{b_1} \rangle$ steps to $\langle S_b; e_{b_2} \rangle$. Since the only way an error configuration can arise is by the E-Put-Err rule, we can assume in all other cases that $\sigma_b \neq \text{error}$.

(a) Case E-Beta: By symmetry with case 1h.

(b) Case E-New: By symmetry with case 2h.

(c) Case E-Put: By symmetry with case 3h.

(d) Case E-Put-Err: By symmetry with case 4h.

(e) Case E-Get: By symmetry with case 5h.
(f) Case E-Freeze-Init: By symmetry with case 6h.

(g) Case E-Spawn-Handler: By symmetry with case 7h.

(h) Case E-Freeze-Final: We have $S_a = S[l \mapsto (d_1, \text{true})]$ and $S_b = S[l' \mapsto (d'_1, \text{true})]$.

Now consider whether $l = l'$:

- If $l \neq l'$:
  
  TODO:

- If $l = l'$:
  
  Note that since $l = l'$, $d_1 = d'_1$.

  TODO:

(i) Case E-Freeze-Simple: Similar to case 8h, since $S_a = S[l \mapsto (d_1, \text{true})]$ and $S_b = S[l' \mapsto (d'_1, \text{true})]$.

(9) Case E-Freeze-Simple: We have $S_a = S[l \mapsto (d_1, \text{true})]$.

(a) Case E-Beta: By symmetry with case 1i.

(b) Case E-New: By symmetry with case 2i.

(c) Case E-Put: By symmetry with case 3i.

(d) Case E-Put-Err: By symmetry with case 4i.

(e) Case E-Get: By symmetry with case 5i.

(f) Case E-Freeze-Init: By symmetry with case 6i.

(g) Case E-Spawn-Handler: By symmetry with case 7i.

(h) Case E-Freeze-Final: By symmetry with case 8i.

(i) Case E-Freeze-Simple: Similar to case 9h, since $S_a = S[l \mapsto (d_1, \text{true})]$ and $S_b = S[l' \mapsto (d'_1, \text{true})]$.

□
A. PROOFS

A.18. Proof of Lemma 3.11

Proof. Suppose $\sigma \mapsto \sigma'$ and $\sigma \mapsto^m \sigma''$, where $1 \leq m$. We are required to show that either:

1. there exist $\sigma_c, i, j, \pi$ such that $\sigma' \mapsto^i \sigma_c$ and $\pi(\sigma'') \mapsto^j \sigma_c$ and $i \leq m$ and $j \leq 1$, or
2. there exists $k \leq m$ such that $\sigma' \mapsto^k \text{error}$, or there exists $k \leq 1$ such that $\sigma'' \mapsto^k \text{error}$.

We proceed by induction on $m$. In the base case of $m = 1$, the result is immediate from Lemma 3.10, with $k = 1$.

For the induction step, suppose $\sigma \mapsto^m \sigma'' \mapsto \sigma'''$ and suppose the lemma holds for $m$.

We show that it holds for $m + 1$, as follows.

From the induction hypothesis, we have that either:

1. there exist $\sigma_c', i', j', \pi'$ such that $\sigma' \mapsto^i \sigma_c'$ and $\pi'(\sigma'') \mapsto^{j'} \sigma_c'$ and $i' \leq m$ and $j' \leq 1$, or
2. there exists $k' \leq m$ such that $\sigma' \mapsto^{k'} \text{error}$, or there exists $k' \leq 1$ such that $\sigma'' \mapsto^{k'} \text{error}$.

We consider these two cases in turn:

1. There exist $\sigma_c', i', j', \pi'$ such that $\sigma' \mapsto^{i'} \sigma_c'$ and $\pi'(\sigma'') \mapsto^{j'} \sigma_c'$ and $i' \leq m$ and $j' \leq 1$:

   We proceed by cases on $j'$:
   - If $j' = 0$, then $\pi'(\sigma'') = \sigma_c'$.
     Since $\sigma'' \mapsto \sigma'''$, we have that $\pi'(\sigma'') \mapsto \pi'(\sigma''')$ by Lemma 3.3 (Permutability).
     We can then choose $\sigma_c = \pi'(\sigma''')$ and $i = i' + 1$ and $j = 0$ and $\pi = \pi'$. The key is that
     $\sigma' \mapsto^{i'} \sigma_c' = \pi'(\sigma'') \mapsto^{j'} \pi'(\sigma''')$ for a total of $i' + 1$ steps.
   - If $j' = 1$:
     First, since $\pi'(\sigma'') \mapsto^{j'} \sigma_c'$, then by Lemma 3.3 (Permutability) we have that $\sigma'' \mapsto^{j'} \pi^{j'-1}(\sigma_c')$.
     Then, by $\sigma'' \mapsto^{j'} \pi^{j'-1}(\sigma_c')$ and $\sigma'' \mapsto \sigma'''$ and Lemma 3.10, one of the following two cases is true:
(a) There exist \( \sigma'' \) and \( i'' \) and \( j'' \) and \( \pi'' \) such that \( \pi''^{-1}(\sigma'_c) \rightarrow i'' \sigma'' \) and \( \pi''(\sigma''') \rightarrow j'' \sigma'_c \) and \( i'' \leq 1 \) and \( j'' \leq 1 \).

Since \( \pi''^{-1}(\sigma'_c) \rightarrow i'' \sigma'' \), by Lemma 3.3 (Permutability) we have that \( \sigma'_c \rightarrow i'' \pi'(\sigma'''). \)

So we also have \( \sigma'_c \rightarrow i'' \sigma'' \).

Since \( \pi''(\sigma''') \rightarrow j'' \sigma'_c \), by Lemma 3.3 (Permutability) we have that \( \pi'(\pi''(\sigma''')) \rightarrow j'' \sigma'_c \).

In summary, we pick \( \sigma'_c = \pi'(\sigma''') \) and \( i = i' + i'' \) and \( j = j'' \) and \( \pi = \pi'' \circ \pi' \), which is sufficient because \( i = i' + i'' \leq m + 1 \) and \( j = j'' \leq 1 \).

(b) \( \pi''^{-1}(\sigma'_c) \rightarrow \text{error} \) or \( \sigma'' \rightarrow \text{error} \).

If \( \sigma'' \rightarrow \text{error} \), then choosing \( k = 1 \) satisfies the proof.

Otherwise, \( \pi''^{-1}(\sigma'_c) \rightarrow \text{error} \). Then, by Lemma 3.5 we have that \( \sigma'_c \rightarrow \pi'(\text{error}) \).

By Definition 3.11, \( \pi'(\text{error}) = \text{error} \), and so \( \sigma'_c \rightarrow \text{error} \). Therefore \( \sigma'_c \rightarrow i' \sigma'_c \rightarrow \text{error} \).

Hence \( \sigma'_c \rightarrow i'+1 \text{error} \).

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 \( \sigma' \rightarrow k' \text{error} \), or there exists \( k' \leq 1 \) such that \( \sigma'' \rightarrow k' \text{error} \): 

If there exists \( k' \leq m \) such that \( \sigma' \rightarrow k' \text{error} \), then choosing \( k = k' \) satisfies the proof.

Otherwise, there exists \( k' \leq 1 \) such that \( \sigma'' \rightarrow k' \text{error} \). We proceed by cases on \( k' \):

- If \( k' = 0 \), then \( \sigma'' = \text{error} \).

  Hence this case is not possible, since \( \sigma'' \rightarrow \sigma''' \) and \( \text{error} \) cannot step.

- If \( k' = 1 \):

  From \( \sigma'' \rightarrow \sigma''' \) and \( \sigma'' \rightarrow k' \text{error} \) and Lemma 3.10, one of the following two cases is true:

  (a) There exist \( \sigma''_c \) and \( i'' \) and \( j'' \) and \( \pi'' \) such that \( \text{error} \rightarrow i'' \sigma''_c \) and \( \pi''(\sigma''') \rightarrow j'' \sigma''_c \) and \( i'' \leq 1 \) and \( j'' \leq 1 \).

  Since \( \text{error} \) cannot step, \( i'' = 0 \) and \( \sigma''_c = \text{error} \).
A. PROOFS

By Definition 3.11, \( \pi''(\sigma''') = \sigma''' \).

Hence \( \sigma''' \rightarrow \sigma'' \text{ error} \).

LK: This is the one place that we need to allow \( k \) to be \( \leq 1 \) instead of exactly 1.

Since \( j'' \leq 1 \), choosing \( k = j'' \) satisfies the proof.

(b) \( \text{error} \rightarrow \text{error} \) or \( \sigma''' \rightarrow \text{error} \).

Since \( \text{error} \) cannot step, \( \sigma''' \rightarrow \text{error} \).

Hence choosing \( k = 1 \) satisfies the proof.

\[ \square \]


Proof. Suppose that \( \sigma \rightarrow^n \sigma' \) and \( \sigma \rightarrow^m \sigma'' \), where \( 1 \leq n \) and \( 1 \leq m \). We are required to show that either:

1. there exist \( \sigma_c, i, j, \pi \) such that \( \sigma' \rightarrow^i \sigma_c \) and \( \pi(\sigma''') \rightarrow^j \sigma_c \) and \( i \leq m \) and \( j \leq n \), or
2. there exists \( k \leq m \) such that \( \sigma' \rightarrow^k \text{error} \), or there exists \( k \leq n \) such that \( \sigma'' \rightarrow^k \text{error} \).

We proceed by induction on \( n \). In the base case of \( n = 1 \), the result is immediate from Lemma 3.11.

For the induction step, suppose \( \sigma \rightarrow^n \sigma' \rightarrow \sigma'' \) and suppose the lemma holds for \( n \).

We show that it holds for \( n + 1 \), as follows.

From the induction hypothesis, we have that either:

1. there exist \( \sigma'_c, i', j', \pi' \) such that \( \sigma' \rightarrow^{i'} \sigma'_c \) and \( \pi'(\sigma''') \rightarrow^{j'} \sigma'_c \) and \( i' \leq m \) and \( j' \leq n \), or
2. there exists \( k' \leq m \) such that \( \sigma' \rightarrow^{k'} \text{error} \), or there exists \( k' \leq n \) such that \( \sigma'' \rightarrow^{k'} \text{error} \).

We consider these two cases in turn:

1. there exist \( \sigma'_c, i', j', \pi' \) such that \( \sigma' \rightarrow^{i'} \sigma'_c \) and \( \pi'(\sigma''') \rightarrow^{j'} \sigma'_c \) 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 \( \pi' (\sigma'') \twoheadrightarrow j' \ \sigma'_c \twoheadrightarrow \sigma''' \), and \( j' + 1 \leq n + 1 \) since \( j' \leq n \), the case is satisfied.

• If \( i' \geq 1 \):

From \( \sigma' \twoheadrightarrow \sigma''' \) and \( \sigma' \twoheadrightarrow i' \ \sigma'_c \) and Lemma 3.11, one of the following two cases is true:

(a) There exist \( \sigma''_c \) and \( i'' \) and \( j'' \) and \( \pi'' \) such that \( \sigma'' \twoheadrightarrow i'' \ \sigma''_c \) and \( \pi'' (\sigma'_c) \twoheadrightarrow j'' \ \sigma''_c \) and \( i'' \leq i' \) and \( j'' \leq 1 \).

Since \( \pi' (\sigma'') \twoheadrightarrow j'' \ \sigma'_c \), by Lemma 3.3 (Permutability) we have that \( \pi'' (\pi' (\sigma'')) \twoheadrightarrow j'' \ \sigma''_c \).

So we also have \( \pi'' (\pi' (\sigma'')) \twoheadrightarrow j'' \ \pi'' (\sigma'_c) \twoheadrightarrow j'' \ \sigma''_c \).

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 \( j = j' + j'' \leq n + 1 \).

(b) There exists \( k'' \leq i' \) such that \( \sigma''' \twoheadrightarrow i'' \ \sigma'_c \) error, or there exists \( k'' \leq 1 \) such that \( \sigma'_c \twoheadrightarrow k'' \ \sigma'_c \) error.

If there exists \( k'' \leq i' \) such that \( \sigma''' \twoheadrightarrow k'' \ \sigma'_c \) error, then choosing \( k = k'' \) satisfies the proof, since \( k'' \leq i' \leq m \).

Otherwise, there exists \( k'' \leq 1 \) such that \( \sigma'_c \twoheadrightarrow k'' \ \sigma'_c \) error.

Hence by Lemma 3.3 (Permutability), we have that \( \pi' (\sigma'_c) \twoheadrightarrow k'' \ \pi' (\sigma'_c) \) error.

By Definition 3.11, \( \pi' (\sigma'_c) \) = error. Hence \( \pi' (\sigma'_c) \twoheadrightarrow k'' \ \sigma'_c \) error.

Since \( \pi' (\sigma'') \twoheadrightarrow j'' \ \sigma'_c \), by Lemma 3.3 (Permutability), we have that \( \sigma'' \twoheadrightarrow j'' \ \pi' (\sigma'_c) \).

Therefore, \( \sigma'' \twoheadrightarrow j'' \ \pi' (\sigma'_c) \twoheadrightarrow k'' \ \sigma'_c \) error.

Hence \( \sigma'' \twoheadrightarrow j'' + k'' \ \sigma'_c \) 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 \( \sigma' \twoheadrightarrow k' \ \sigma'_c \) error, or there exists \( k' \leq n \) such that \( \sigma'' \twoheadrightarrow k' \ \sigma'_c \) error:

If there exists \( k' \leq n \) such that \( \sigma'' \twoheadrightarrow k' \ \sigma'_c \) error, then choosing \( k = k' \) satisfies the proof.

Otherwise, there exists \( k' \leq m \) such that \( \sigma' \twoheadrightarrow k' \ \sigma'_c \) error. We proceed by cases on \( k' \):
A. PROOFS

- If \( k' = 0 \), then \( \sigma' = \text{error} \).

Hence this case is not possible, since \( \sigma' \rightarrow \sigma''' \) and \( \text{error} \) cannot step.

- If \( k' \geq 1 \):

  From \( \sigma' \rightarrow \sigma''' \) and \( \sigma' \rightarrow^k \text{error} \) and Lemma 3.11, one of the following two cases is true:
  
  (a) There exist \( i'' \) and \( j'' \) and \( \pi'' \) such that \( \sigma'' \rightarrow i'' \sigma_c'' \) and \( \pi''(\text{error}) \rightarrow j'' \sigma_c'' \) and \( i'' \leq k' \) and \( j'' \leq 1 \).

  By Definition 3.11, \( \pi''(\text{error}) = \text{error} \). Hence \( \text{error} \rightarrow j'' \sigma_c'' \).

  Since \( \text{error} \) cannot step, \( j'' = 0 \) and \( \sigma'' = \text{error} \).

  Hence \( \sigma'' \rightarrow \text{error} \).

  Since \( i'' \leq k' \leq m \), choosing \( k = i'' \) satisfies the proof.

  (b) There exists \( k'' \leq k' \) such that \( \sigma''' \rightarrow^k \text{error} \), or there exists \( k'' \leq 1 \) such that \( \text{error} \rightarrow^k \text{error} \).

  Since \( \text{error} \) cannot step, there exists \( k'' \leq k' \) such that \( \sigma''' \rightarrow^k \text{error} \).

  Since \( k'' \leq k' \leq m \), choosing \( k = k'' \) satisfies the proof.

\[ \Box \]

A.20. Proof of Theorem 5.2

Proof. Consider replica \( i \) of a threshold CvRDT \((S, \leq, s^0, q, t, u, m)\). Let \( S \) be a threshold set with respect to \((S, \leq)\). Consider a method execution \( t_i^{k+1}(S) \) (i.e., a threshold query that is the \( k + 1 \)th method execution on replica \( i \), with threshold set \( S \) as its argument) that returns some set of activation states \( S_a \in S \).

For part 1 of the theorem, we have to show that threshold queries with \( S \) 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 \( t^{k'}_i(S) \) on \( i \) returns \( S_a \).
Since $t_i^{k+1}(S)$ 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, $t_i^{k'}(S)$ 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 $t_j^{x+1}(S)$ 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 $t_j^{x+1+i}(S)$ returns block, and
- for all $i \geq n$, the threshold query $t_j^{x+1+i}(S)$ returns $S_a$.

Consider arbitrary $x \geq 0$. Recall that $s_j^x$ is the state of replica $j$ after the $x$th method execution, and therefore $s_j^x$ is also the state of $j$ when $t_j^{x+1}(S)$ runs. We have three cases to consider:

- $s_i^k \leq s_j^x$. (That is, replica $i$’s state after the $k$th method execution on $i$ is at or below replica $j$’s state after the $x$th method execution on $j$.) Choose $n = 0$. We have to show that, for all $i \geq n$, the threshold query $t_j^{x+1+i}(S)$ returns $S_a$. Since $t_i^{k+1}(S)$ returns $S_a$, we know that there exists an $s_a \in S_a$ 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, $t_j^{x+1}(S)$ returns $S_a$. Then, by part 1 of the theorem, we have that subsequent executions $t_j^{x+1+i}(S)$ 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 $k$th method execution on $i$ is above replica $j$’s state after the $x$th 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 $t_j^{x+1+i}(S)$ returns $S_a$. Since $s_a' \leq s_j^x$,

¹The 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 $i$. DRAFT: March 14, 2015 230
by Definition 5.4, \( t_j^{x+1}(S) \) returns \( S_a \). Then, by part 1 of the theorem, we have that subsequent executions \( t_j^{x+1+i}(S) \) 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 \( s'_a \leq s^x_j \). Since \( t_i^{k+1}(S) \) 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 \( s^x_j \) to \( s^k_i \).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 \( s^x_j \) to a state at or above some \( s'_a \) must reach replica \( j \).

Let the \( x + 1 + r \)th method execution on \( j \) be the first update on \( j \) that updates its state to some \( s^x_j + r \geq s'_a \), 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 \( t_j^{x+1+i}(S) \) returns block, and that for all \( i \geq r + 1 \), the threshold query \( t_j^{x+1+i}(S) \) 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 \( s^x_j + r \geq s'_a \), we have by Definition 5.4 that for all \( i \) in the range \( 0 \leq i \leq r \), the threshold query \( t_j^{x+1+i}(S) \) returns block.

For the latter, since \( s^x_j + r \geq s'_a \), by Definition 5.4 we have that \( t_j^{x+1+r+1}(S) \) returns \( S_a \), and by part 1 of the theorem, we have that for \( i \geq r + 1 \), subsequent executions \( t_j^{x+1+i}(S) \) at replica \( j \) will also return \( S_a \), and so the case holds.

- \( s^k_i \not\leq s^x_j \) and \( s^x_j \not\leq s^k_i \). (That is, replica \( i \)'s state after the \( k \)th method execution on \( i \) is not comparable to replica \( j \)'s state after the \( x \)th method execution on \( j \).) Similar to the previous case.

²We know that \( i \)'s state was once at or below \( s^x_j \), because \( i \) and \( j \) started at the same state \( s^0 \) and can both only grow. Hence the least that \( s^x_j \) can be is \( s^0 \), and we know that \( i \) was originally \( s^0 \) as well.

³We 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.
APPENDIX B

PLT Redex Models of $\lambda_{LVar}$ and $\lambda_{LVish}$

TODO: Edit this text.

We have developed a runnable version of the LVish calculus\textsuperscript{1} using the PLT Redex semantics engineering toolkit [21]. In the Redex of today, it is not possible to directly parameterize a language definition by a lattice.\textsuperscript{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 $\lambda_{LVish}$, 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 lub, one writes:

\begin{verbatim}
(define-LVish-language nat downset-op max natural)
\end{verbatim}

\textsuperscript{1}Available at \url{http://github.com/iu-parfunc/lvars}.

\textsuperscript{2}See discussion at \url{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 $L_{Var}$ and $L_{Vish}$ and add them here.